

CHAPTER 18 SQL Optimization

Oracle SQL By Example, Fourth Edition by Alice Rischert. Published by Prentice Hall. Copyright © 2008 by Pearson Education, Inc.

CHAPTER OBJECTIVES

In this chapter, you will learn about:

- ▶ The Oracle Optimizer
- ▶ Writing Effective SQL Statements

Throughout this book, you have seen alternate SQL statements for many solutions. This chapter focuses on helping you determine which SQL statement will most efficiently and quickly return results. It provides an overview of the workings of the Oracle optimizer and describes SQL performance tuning techniques. The list of tuning suggestions in this chapter is by no means comprehensive but merely a starting point. When you understand how to read the execution steps of SQL statements, you will have gained a better understanding of Oracle's optimization strategies and will be able to focus on tuning problem areas with alternate SQL statements and techniques.

LAB 18.1 The Oracle Optimizer and Writing Effective SQL Statements

LAB OBJECTIVES

After this lab, you will be able to:

- ▶ Read an Execution Plan
- ▶ Understand Join Operations
- ▶ Use Alternate SQL Statements

Poor system performance is often caused by one or a combination of problems: poor database design, improper tuning of the Oracle server, and poorly written SQL statements. A well-thought-out database design has the greatest positive impact on database performance, followed by effectively written SQL statements, and then tuning the Oracle server itself. This chapter focuses on writing effective SQL statements.

The Oracle database server provides a number of tools that help you improve the efficiency of SQL statements. This chapter shows you how to obtain an *execution plan*,

which is a sequence of the steps that Oracle carries out to perform a specific SQL command. You can change the execution plan by choosing an alternate SQL statement or by adding a *hint*, which is a directive to execute the statement differently.

Using hints, you can force the use of a specific index, change the join order, or change join method. Before you learn more about the execution plan and the query tuning tools, you must understand the basics of how the Oracle database server evaluates a SQL statement.

SQL Statement Processing

Before a SQL statement returns a result set, the server performs a number of operations that are completely transparent to the user. First, Oracle creates a cursor, an area in memory where Oracle stores the SQL statement and all associated information. Next, Oracle parses the SQL statement. This entails checking the syntax for validity, checking whether all the column and table names exist, and determining whether the user has permission to access these tables and columns. Part of the parsing operation is also the determination of the execution plan.

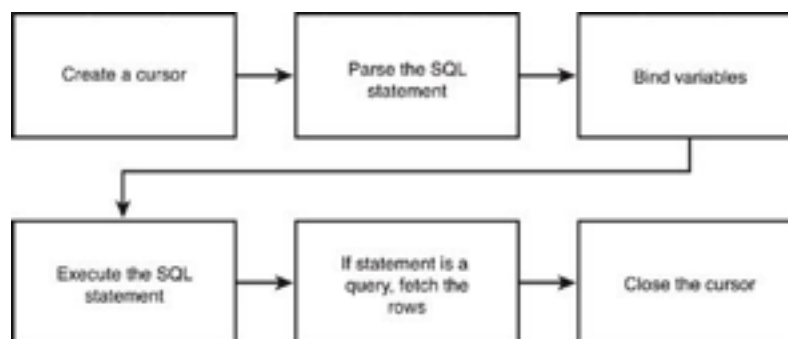
804

805

Because parsing requires time and resources, there are ways to eliminate parsing when repeatedly executing similar statements. Under these circumstances, the use of bind variables may be helpful. Bind variables are placeholders for values, which get substituted with different values. Oracle maintains a cache of recently executed SQL statements and their respective execution plan, so if an identical statement has been used previously, it does not need to be reparsed.

If bind variables are being used, they are associated with the appropriate values. Next, the SQL statement is executed. If the SQL statement is a query, the result needs to be fetched. Once all the rows are fetched, the cursor is closed. [Figure 18.1](#) shows an overview of these steps.

FIGURE 18.1 Overview of SQL statement processing steps



The Oracle Optimizer

The Oracle optimizer, which is part of the Oracle software, examines each SQL statement and chooses the best execution plan for it. An execution plan consists of a sequence of steps that are necessary to execute the SQL statement.

The rule-based optimizer (RBO) was Oracle's first optimizer, employed by Oracle since its beginnings. The RBO determined the execution plan through a number of rigid rules. This optimizer is no longer supported by Oracle, however, and it has been replaced by the cost-based optimizer (CBO).

The CBO considers statistics when determining the best plan, and these statistics are stored in the data dictionary. Statistics include values such as the number of rows in the table and the selectivity of columns, among many other factors. Statistics are gathered automatically in Oracle if they become stale. In versions prior to Oracle 10g, you had to perform this step manually by using the DBMS_STATS package or the ANALYZE command. The statistics help Oracle by calculating the estimated cost of the various execution options to help determine the best and lowest-cost plan.

805

806

Choosing the Optimizer Mode

The CBO has a number of optimizer modes. You can decide whether you want to optimize for best overall throughput (ALL_ROWS) or best response (FIRST_ROWS_*n*). The ALL_ROWS mode is most useful for reporting and batch processing, whereas FIRST_ROWS_*n* is best for interactive applications, to quickly retrieve the first *n* rows. Based on the chosen optimizer mode, the gathered statistics, and certain database initialization parameters, the Oracle optimizer chooses the best execution plan. [Table 18.1](#) lists the optimizer modes.

You modify the optimizer mode on the database instance level in the database initialization file with the OPTIMIZER_MODE parameter. Or, for an entire session, you use the ALTER SESSION SET OPTIMIZER_MODE command. For an individual SQL statement, you can override it with a hint.

TABLE 18.1 Optimizer Mode

OPTIMIZER MODE	EXPLANATION
FIRST_ROWS_n	This mode has the goal of retrieving the specified first n row(s). This mode is best for interactive programs that need to display n initial rows of data on a screen. (This setting differs from the FIRST_ROWS mode in that you list the number of rows. The FIRST_ROWS optimizer goal, without the specified number of rows, is supported for backward compatibility.)
ALL_ROWS	The goal of this mode is to retrieve all the rows and achieve the best performance with minimal resources. This is typically the default setting.

Keeping Statistics Up-to-Date in Oracle

Accurate statistics about the distribution of data are essential for good performance. Oracle automatically gathers statistics for objects that have stale or missing statistics and periodically updates these statistics. The statistics are stored in the data dictionary. Gathering of statistics is accomplished through a job called BSLN_MAINTAIN_STATS_JOB, which is created

automatically when the database is initially created. Oracle's internal job scheduler periodically runs this job to check for stale or missing statistics. If you have access to the DBA_ dictionary views, you can check whether the job is running by executing the following statement.

```
SELECT job_name, enabled,  
       last_start_date  
       FROM dba_scheduler_jobs  
WHERE job_name =  
       'BSLN_MAINTAIN_STATS_JOB'
```

806

807

JOB_NAME	ENABL	LAST_START_DATE
BSLN_MAINTAIN_STATS_JOB	TRUE	31-JAN-09 10.00.02.109124 PM -05:00

1 row selected.

If you create a new index on a table or rebuild an existing one, Oracle automatically collects the statistics. However, the statistics may be stale when 10 percent or more of the rows change due to INSERT, UPDATE, or DELETE operations. Oracle keeps track of the modifications in the USER_TAB_MODIFICATIONS data dictionary.

The USER_TAB_MODIFICATIONS data dictionary view shows the tables owned by the current user that are being monitored. Furthermore, it shows the volume of data modified since the last gathering of statistics. The

number found in the INSERTS, UPDATES, and DELETES columns of the view is approximate; these columns may not be populated for a few hours after the completion of the *Data Manipulation Language* (DML) operation.

```
SELECT table_name, inserts, updates, deletes,
       timestamp
FROM   user_tab_modifications
```

TABLE_NAME	INSERTS	UPDATES	DELETES	TIMESTAMP
COURSE	1	0	0	30-JAN-09

1 row selected.

Gathering Statistics Manually

In case you want to manually gather statistics (which may be useful after a large data load or mass update), the following paragraphs explain some of the involved procedures you will need to execute. You can check for the presence of statistics by querying the data dictionary views `ALL_TABLES` and `ALL_INDEXES`. After statistics about a table or an index are gathered, a number of columns in these data dictionary views contain values. For example, in the `USER_TABLES` view, the `NUM_ROWS` column contains the number of rows in the table, the average row length, and the date and time the statistics were last gathered.

```
SELECT table_name, num_rows, avg_row_len,
       TO_CHAR(last_analyzed, 'MM/DD/YYYY')
       AS last_analyzed
FROM user_tables
```

TABLE_NAME	NUM_ROWS	AVG_ROW_LEN	LAST_ANALY
-----	-----	-----	-----
INSTRUCTOR	10	85	01/23/2009
GRADE	2004	49	01/23/2009
...			
ZIPCODE	227	53	01/23/2009

10 rows selected.

807

808

For indexes, you can review the number of rows, the number of distinct keys, and the date and time the statistics were last updated. Other related data dictionary views contain additional information, but querying USER_TABLES and USER_INDEXES provides you with some of the most essential information.

```
SELECT index_name, num_rows, distinct_keys,
       TO_CHAR(last_analyzed, 'MM/DD/YYYY')
       AS last_analyzed
FROM user_indexes
```

INDEX_NAME	NUM_ROWS	DISTINCT_KEYS	LAST_ANALY
-----	-----	-----	-----
INST_ZIP_FK_I	9	4	01/23/200
GR_GRTW_FK_I	2004	252	011/23/2009
...			
CRSE_PK	30	30	011/23/2009

20 rows selected.

THE DBMS_STATS PACKAGE

The DBMS_STATS package is an Oracle-supplied PL/SQL package that generates and manages statistics for use by the CBO. [Table 18.2](#) lists a few of the many procedures in the package for gathering statistics.

TABLE 18.2 DBMS_STATS Procedures That Gather Statistics

PROCEDURE	PURPOSE
GATHER_TABLE_STATS	Gathers table, column, and index statistics.
GATHER_INDEX_STATS	Gathers index statistics.
GATHER_SCHEMA_STATS	Gathers statistics for all objects in a schema.
GATHER_DATABASE_STATS	Gathers statistics for all objects in a database instance.
GATHER_SYSTEM_STATS	Gathers system statistics about the CPU and I/O.

Following are some examples of how to execute the procedures to collect statistics. The following statement gathers exact statistics for the COURSE table, located in the STUDENT schema. You can execute the PL/SQL

procedure in SQL*Plus or SQL Developer via the Run Script icon.

```
EXEC DBMS_STATS.GATHER_TABLE_STATS (
    ownname=>'STUDENT',
    tabname=>'COURSE' );
```

PL/SQL procedure successfully completed.

The two parameters OWNNAME for the schema name and TABNAME for the table name are required. The example uses the named notation syntax (ownname=>) to identify each parameter with the appropriate value. You do not need to list the parameter names OWNNAME and TABLENAME if you supply the parameter values in the order in which they are defined in the package.

```
EXEC
DBMS_STATS.GATHER_TABLE_STATS ('STUD
ENT', 'COURSE' );
```

The procedure has additional parameters (for example, ESTIMATE_PERCENT) that let you specify the sample percentage. If you don't specify the additional parameters, default values are assigned. In the previous examples, ESTIMATE_PERCENT is not specified, so the default is to compute the exact statistics.

808

809

If the different parameters do not fit on one line or if you want to separate the parameters, write the procedure call as follows:

```
BEGIN
  DBMS_STATS.GATHER_TABLE_STATS (
    ownname=>'STUDENT',
    tabname=>'COURSE',
    cascade=>TRUE) ;
END;
/
```

The procedure adds a third parameter, the CASCADE parameter, and sets it to TRUE. This instructs the procedure to collect both table and index statistics simultaneously. To find out the different parameter names of a procedure, you issue the SQL*Plus DESCRIBE command in SQL*Plus. It lists all the individual procedures available and their respective parameters. The In/Out column indicates whether the procedure requires an input parameter or returns an output value. You can find out more information about each parameter and the respective default values in the *Oracle 11g PL/SQL Packages and Types References* manual.

```
SQL> DESCRIBE DBMS_STATS
```

```
SQL> DESCRIBE DBMS_STATS
...
PROCEDURE GATHER_TABLE_STATS
Argument Name          Type          In/Out Default?
-----
OWNNAME                VARCHAR2      IN
TABNAME                VARCHAR2      IN
PARTNAME               VARCHAR2      IN      DEFAULT
ESTIMATE_PERCENT       NUMBER        IN      DEFAULT
BLOCK_SAMPLE           BOOLEAN        IN      DEFAULT
METHOD_OPT             VARCHAR2      IN      DEFAULT
DEGREE                 NUMBER        IN      DEFAULT
GRANULARITY            VARCHAR2      IN      DEFAULT
CASCADE                BOOLEAN        IN      DEFAULT
...
```

EXACT STATISTICS OR SAMPLE SIZE

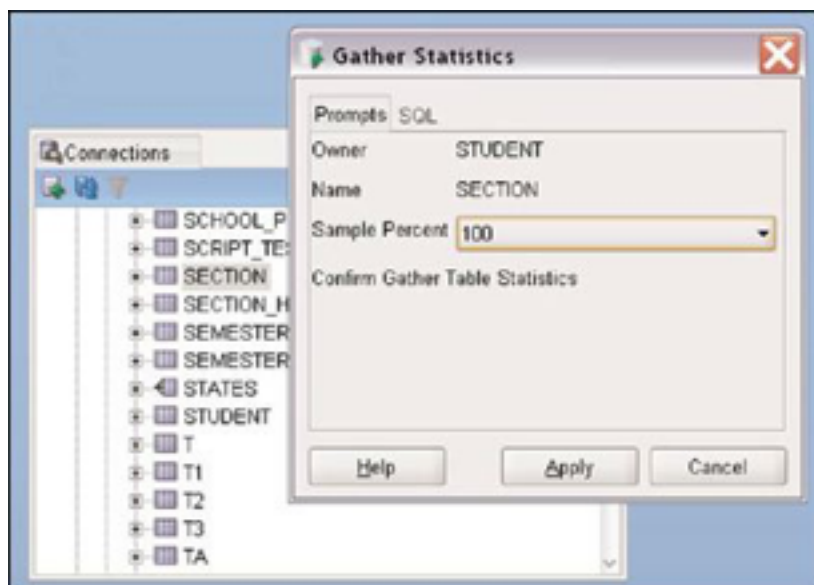
As mentioned earlier, statistics can be either computed exactly or estimated. Because calculating exact statistics may take a long time on very large tables, it is sometimes more practical to estimate sufficiently accurate statistics. For large tables, a sampling percentage size of 10 to 25 percent is often sufficient, or you can let Oracle determine the size of the sample.

809

USING SQL DEVELOPER TO GATHER STATISTICS

The SQL Developer menu options make it easy to gather statistics (see [Figure 18.2](#)). By right-clicking on a table's node, you will be able to choose Statistics menu. From there, you can choose Gather Statistics option, which evokes the menu in [Figure 18.2](#) where you choose the sample size and shows in the SQL tab the corresponding DBMS_STATS.GATHER_TABLE_STATS procedure to be executed. If you click a connection name instead, you can also gather statistics for the schema.

FIGURE 18.2 Gathering statistics in SQL Developer



Managing Statistics

Besides gathering statistics, the DBMS_STATS package includes procedures to modify, view, export, import, delete, lock, and restore statistics. For example, you can save the current statistics before gathering new statistics so you can restore them if the performance of the system is adversely affected. You can also copy the statistics from one database instance to another. This is useful if you want to test how your SQL statements behave in a different environment (for example, test vs. production). You can also lock the statistics so they remain unchanged. [Table 18.3](#) shows a few of the DBMS_STATS procedures that manage statistics for an individual table.

TABLE 18.3 DBMS_STATS Procedures

PROCEDURE	PURPOSE
DELETE_TABLE_STATS	Deletes statistics for an individual table.
CREATE_STAT_TABLE	Creates a table to hold statistics for import/export.
LOCK_TABLE_STATS	Freezes the current statistics, including indexes for an individual table.
PROCEDURE	PURPOSE
UNLOCK_TABLE_STATS	Unlocks the table and index statistics.
RESTORE_SCHEMA_STATS	Restores all the statistics for a specified schema for a particular timestamp. This is useful if performance degrades and you want to restore the previous set of statistics.
EXPORT_TABLE_STATS	Exports statistics about an individual table so it can be used for a later import.
IMPORT_TABLE_STATS	Imports table statistics into the data dictionary.

810

811

Timing the Execution of a Statement

If a SQL statement does not perform well, you need a baseline to compare the execution time of other alternative SQL statements.



Repeated executions of the same or similar statements take less time than the initial execution because the data no longer needs to be retrieved from disk since it is cached in memory. Just because you make a minor change to the statement doesn't mean the statement is actually running faster.

In SQL Developer, you will see the execution time next to the Eraser icon. One simple way to find out the execution duration of a statement within SQL*Plus is to use the SQL*Plus command SET TIMING ON, which returns the elapsed execution time after the execution of the statement completes.

Tuning a SQL statement is effective only if your SQL statement executes against realistic data volumes and column distributions similar to what would be expected

in a production environment. For instance, the execution plan for a join involving two tables varies if the data in the test environment is 100 rows but in production it is 500,000 rows. The Oracle optimizer also evolves with each subsequent version of the Oracle database, so having a test environment that closely resembles your production environment greatly aids in this process.

The Execution Plan

The optimizer creates an *execution plan*, also referred to as an *explain plan*. This plan shows the individual steps the Oracle database executes to process a statement. Oracle reads the execution plan from the inside out, meaning the most indented step is performed first. If two steps have the same level of indentation, the step listed first is the first one executed. The following shows a SQL statement and its execution plan. You'll learn how to obtain such an output shortly. (The ID column on the left is only a number identifying the step; it does not indicate the execution order in any way.)

```
SELECT student_id, last_name
FROM student
WHERE student_id = 123
```

ID	Operation	Name
0	SELECT STATEMENT	
1	TABLE ACCESS BY INDEX ROWID	STUDENT
2	INDEX UNIQUE SCAN	STU_PK

811
812

The first step performed is a lookup of the value 123 in the index STU_PK. Using the index entry, the row is retrieved from the STUDENT table via ROWID, which specifies the location (data file and data block) of the row.

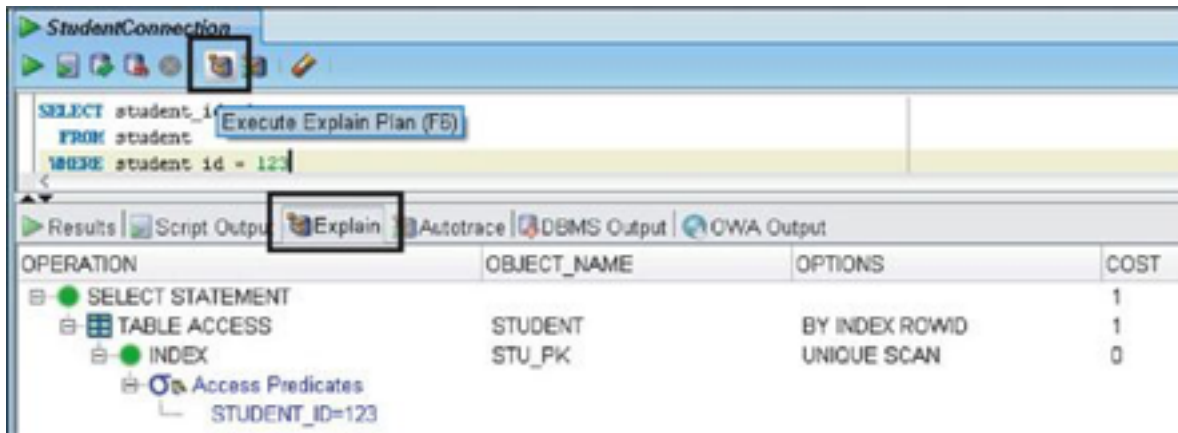
There are various ways to obtain an execution plan. This chapter discusses the use of SQL Developer's Explain Plan and Autotrace features together with the EXPLAIN PLAN FOR command in SQL*Plus, along with the DBMS_XPLAN.DISPLAY package.

The best way to generate an explain plan is by using SQL Developer because it involves only a mouse-click. However, most of the explain plan results in this chapter are shown in the SQL*Plus output format because it is easier to read in the printed format.

SQL DEVELOPER'S EXPLAIN PLAN

SQL Developer allows you to obtain an execution plan by clicking the Execute Explain Plan icon (or pressing F6), as shown in [Figure 18.3](#). The result is graphically displayed in the Explain tab.

FIGURE 18.3 Explain plan output in SQL Developer



SQL DEVELOPER'S AUTOTRACE

In addition to using the Explain Plan icon, another way to generate an explain plan is with the Autotrace icon, located to the right of the Explain Plan icon. The result is shown in the Autotrace tab. In You see an explain plan as well plus you notice additional statistics that are useful for database administrators (DBAs) and advanced SQL users for further fine-tuning of SQL statements. These statistics are present because the statement is actually executed.

812

813



You must have the `SELECT_CATALOG_ROLE` privilege to obtain a result using Autotrace.

DBMS_XPLAN IN SQL*PLUS

You can list the explain plan by using the `EXPLAIN PLAN FOR` command and the `DISPLAY` function of the `DBMS_XPLAN` package. The `DISPLAY` function retrieves the execution plan and runtime statistics, based on the `V$SQL_PLAN` and `V$SQL_PLAN_STATISTICS` data dictionary tables. The following statement shows how you can create an explain plan by using the `EXPLAIN PLAN` command.

```
SQL> EXPLAIN PLAN FOR
      2 SELECT student_id, last_name
      3 FROM student
      4 WHERE student_id = 123
      5 /
```

Explained.

Afterward, you can retrieve the plan by using the following `DISPLAY` function.

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time

0	SELECT STATEMENT		1	12	2 (50)	00:00:01
1	TABLE ACCESS BY INDEX ROWID	STUDENT	1	12	2 (50)	00:00:01
*2	INDEX UNIQUE SCAN	STU_PK	1		1 (100)	00:00:01

If your result does not fit the display of your screen, you can enter the following SQL*Plus commands to increase the line size and suppress any headings.

```
SET LINESIZE 130  
SET PAGESIZE 0
```

813

814



The explain plan display used in this chapter alternates between the SQL*Plus version and the SQL Developer version. One format sometimes lends itself to better readability than the other format within the text of this book. The overall steps in the execution plan do not differ.

Understanding COST, CARDINALITY, ROWS, and BYTES Values

The previous explain plan contains a number of columns that provide more detail about each individual step. The COST value is a number that represents the estimated disk I/O and amount of CPU and memory required to execute the desired action.

The COST helps you determine how involved each step is so you can focus on tuning the steps with the highest cost. The COST is determined using estimated amounts of memory, I/O, and CPU time required to execute the statement, as well as certain Oracle initialization parameters. The number of the ROWS or CARDINALITY column shows how many rows the optimizer expects to process at this step. In SQL*Plus, you can also see the BYTES column displaying the size, in bytes, expected for the step.



While a number of the explain plans in this book do not show the ROWS/CARDINALITY and COST columns for space reasons, you should display them when tuning a statement as these values provide useful details about each individual step.

SQL DEVELOPER OUTPUT PREFERENCES

In SQL Developer, you can move the columns in the Explain Plan and Autotrace tabs into the order you find

most useful. You can also use the Tools/Preference menu to control which of various columns are shown in the various tabs (see [Figure 18.4](#)). The Autotrace column on the right of the screen has additional check boxes for the display.

SQL*PLUS OUTPUT PREFERENCES

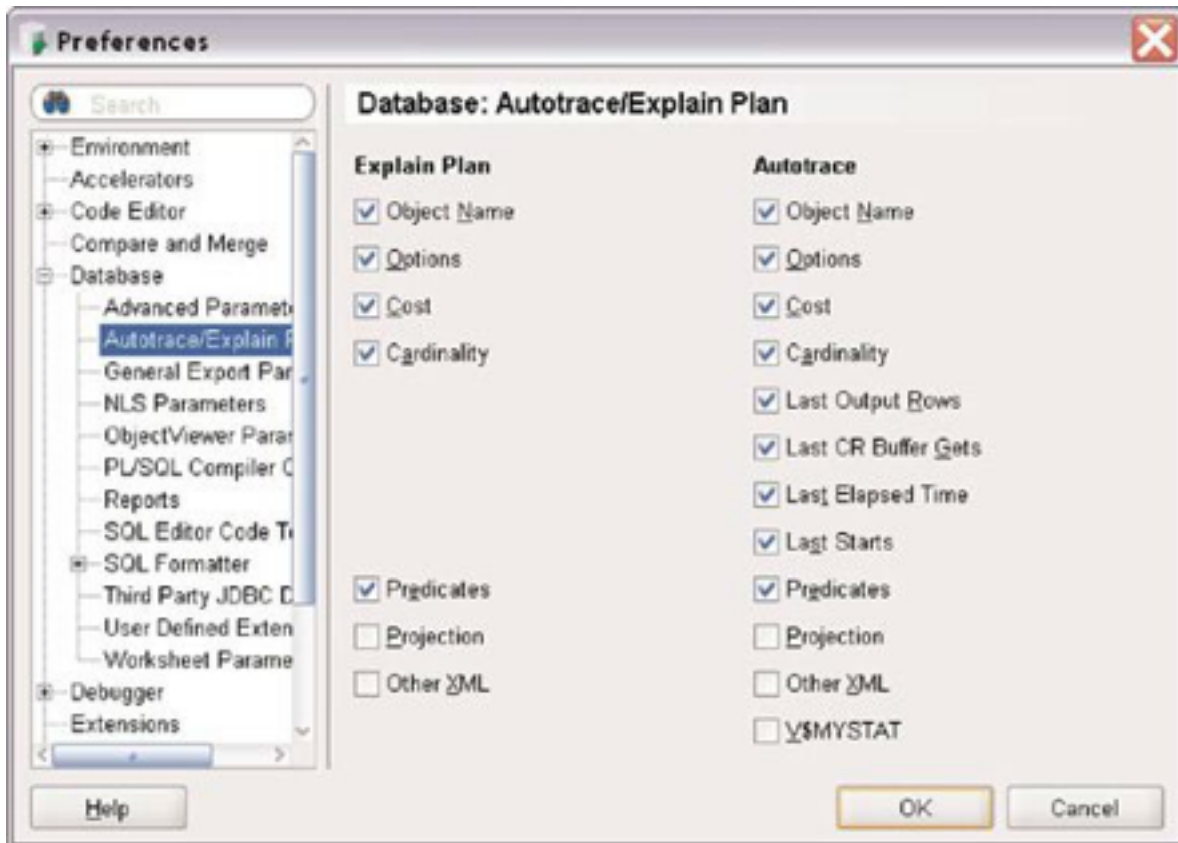
If you execute the DISPLAY function without any parameter, you see the statistics. If you want to display only the minimum plan information, use the BASIC parameter, as in the next statement.

```
SELECT *
FROM
TABLE(DBMS_XPLAN.DISPLAY(null,
null, 'BASIC'))
```

Id	Operation	Name
0	SELECT STATEMENT	
1	TABLE ACCESS BY INDEX ROWID	STUDENT
2	INDEX UNIQUE SCAN	STU_PK

814

FIGURE 18.4 SQL Developer Preferences screen



The DISPLAY function has several parameters, and the syntax is as follows.

```
DBMS_XPLAN.DISPLAY
  (table_name IN VARCHAR2 DEFAULT
  'PLAN_TABLE',
  statement_id IN VARCHAR2 DEFAULT
  NULL,
  format IN VARCHAR2 DEFAULT
  'TYPICAL',
```

```
filter_preds IN VARCHAR2 DEFAULT  
NULL)
```

The first parameter allows you to specify the name of the table where the plan is stored. By default, the explain plan is stored in `PLAN_TABLE`. The second parameter lets you include a `STATEMENT_ID`; this is useful if execute the `EXPLAIN PLAN` command with a `SET STATEMENT_ID` clause to identify different statements. If you do not specify a value, the `DISPLAY` function returns the most recent explained statement. The third parameter permits you to change the display output of the plan. The `BASIC` value shows the operation ID, the operation, and the object name. The `TYPICAL` option is the default and includes the predicate (`WHERE` clause). The `ALL` choice displays all available data, including column information (column projection) and data related to the Oracle parallel server, if applicable. `FILTER_PREDS` indicates whether you want to display the predicate (that is, `WHERE` clause) in the result.

815

816

Hints

If you are not satisfied with the optimizer's plan, you can change it by applying hints. Hints are directives to the optimizer. For example, you can ask to use a particular

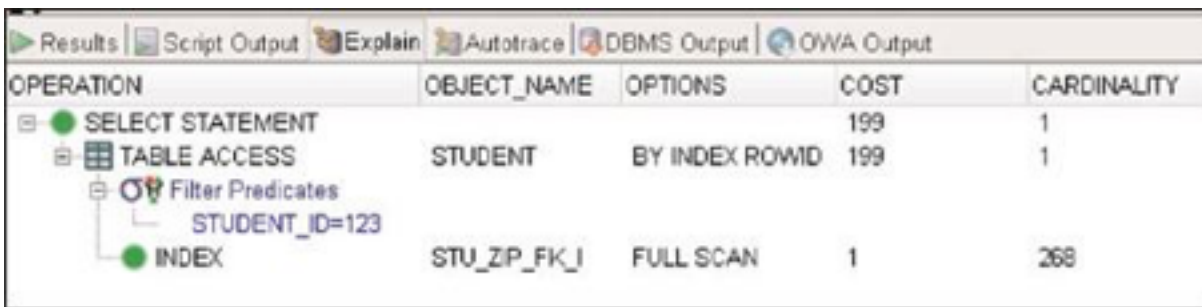
index or to choose a specific join order. Because you know the distribution of the data best, sometimes you can come up with a better execution plan by overriding the default plan with specific hints. In certain instances, this may result in a better plan. For example, if you know that a particular index is more selective for certain queries, you can ask the optimizer to use that index.

A hint is enclosed by either a multiline comment with a plus sign (`/*+ */`) or a single line comment with a plus sign (`--+`). The following statement uses an index hint to scan the `STU_ZIP_FK_I` index on the `STUDENT` table. This index is actually a very poor choice when compared to the `STU_PK` index, but the example demonstrates how you can override the optimizer's default plan.

[Figure 18.5](#) shows a full scan on the specified index for 268 rows before the `WHERE` condition was applied.

```
SELECT /*+ INDEX (student
stu_zip_fk_i) */
       student_id,
       last_name
FROM   student
WHERE  student_id = 123
```

FIGURE 18.5 Forcing a specific index and resulting explain plan result



OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALITY
SELECT STATEMENT			199	1
TABLE ACCESS	STUDENT	BY INDEX ROWID	199	1
Filter Predicates				
STUDENT_ID=123				
INDEX	STU_ZP_FK_I	FULL SCAN	1	268

[Table 18.4](#) lists some of the frequently used hints. You will use some of them in the exercises throughout this lab.

TABLE 18.4 Popular Hints

HINT	PURPOSE
FIRST_ROWS(n)	Returns the first n rows as quickly as possible.
ALL_ROWS	Returns all rows as quickly as possible.
INDEX(tablename indexname)	Uses the specified index. If an alias is used in the FROM clause of the query, be sure to list the alias instead of the table name.
ORDERED	Joins the tables as listed in the FROM clause of the query.
LEADING(tablename)	Specifies which table is the first table in the join order.
USE_MERGE(tablename)	Uses the sort-merge join method to join tables.
USE_HASH(tablename)	Uses the hash join method to join tables.
USE_NL(tablename)	Uses the nested loop join method; the specified table name is the inner table.

816

817

INCORRECTLY SPECIFYING HINTS

If you incorrectly specify a hint, the optimizer ignores it, and you are left to wonder why the hint does not work. The following is an example of the index hint specified incorrectly, and [Figure 18.6](#) shows the resulting explain plan.

```
SELECT /*+ INDEX (student
stu_zip_fk_i) */
      student_id,
      last_name
FROM student s
WHERE student_id = 123
```

FIGURE 18.6 Incorrect hint

OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALITY
SELECT STATEMENT			1	1
TABLE ACCESS	STUDENT	BY INDEX ROWID	1	1
INDEX	STU_PK	UNIQUE SCAN	0	1
Access Predicates STUDENT_ID=123				

Instead of the table name STUDENT, the table alias *s* should be used because an alias is used in the FROM clause of the statement. This incorrect and therefore ignored hint causes the optimizer to use a different, actually more efficient index.

Your hint may also be ignored if you use the `FIRST_ROWS` hint in a query that contains a `GROUP BY` clause, an aggregate function, a set operator, the `DISTINCT` keyword, or an `ORDER BY` clause (if not supported by an index). All these Oracle keywords require that the result or sort first be determined based on all the rows before returning the first row.

Join Types

Determining the type of join and the join order of tables has a significant impact on how efficiently a SQL statement executes. Oracle chooses one of four types of join operations: sort-merge join, hash join, nested-loop join, or cluster join. This lab discusses only the first three, which are the most popular ones.

817

SORT-MERGE JOIN

818

To perform a sort-merge join, a full table scan is executed for each table. In the following SQL statement, the entire `ENROLLMENT` table is read and sorted by the joining column, and then the `STUDENT` table is scanned and sorted. The two results are then merged, and the matching rows are returned for output. The first row is returned only after all the records from both tables are processed.

This join is typically used when the majority of the rows are retrieved, when the join condition is not an equijoin, when no indexes exist on the table to support the join condition, or when a `USE_MERGE` hint is specified (see [Figure 18.7](#)).

```
SELECT /*+ USE_MERGE (e, s) */ *
FROM enrollment e, student s
WHERE s.student_id = e.student_id
```

FIGURE 18.7 Sort-merge join

OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALITY
SELECT STATEMENT			8	226
MERGE JOIN			8	226
SORT		JOIN	4	226
TABLE ACCESS	ENROLLMENT	FULL	3	226
SORT		JOIN	4	268
Access Predicates				
S.STUDENT_ID=E.STUDENT_ID				
Filter Predicates				
S.STUDENT_ID=E.STUDENT_ID				
TABLE ACCESS	STUDENT	FULL	3	268

HASH JOIN

Oracle performs a full table scan on each of the tables and splits each into many partitions in memory. Oracle then builds a hash table from one of these partitions and probes it against the partition of the other table (see [Figure 18.8](#)). A hash join typically outperforms a sort-merge join.

```
SELECT /*+ HASH_JOIN */ *
FROM enrollment e, student s
WHERE s.student_id = e.student_id
```

818

819

FIGURE 18.8 Hash join

OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALITY
SELECT STATEMENT			7	226
HASH JOIN			7	226
Access Predicates				
S.STUDENT_ID=E.STUDENT_ID				
TABLE ACCESS	ENROLLMENT	FULL	3	226
TABLE ACCESS	STUDENT	FULL	3	268

NESTED-LOOP JOIN

With a nested-loop join, the optimizer picks a driving (or outer) table that is the first table in the join chain. In this example, the driving table is the ENROLLMENT table. A full table scan is executed on the driving table and for each row in the ENROLLMENT table; the primary key index of the STUDENT table (here the inner table) is probed to see if the WHERE clause condition is satisfied. If it is, the row is returned in the result set. This probing is repeated until all the rows of the driving table, in this case the ENROLLMENT table, are tested.

In Oracle 11g, the nested-loop join has been further enhanced. Oracle can batch multiple physical reads and

builds a vector. Then the explain plan for a nested loop shows two NESTED LOOPS rows instead of one.

[Figure 18.9](#) displays the execution plan of a nested-loop join with the two NESTED LOOPS rows indicating that the statement is using the enhanced Oracle 11 g join functionality.

```
SELECT /*+ USE_NL(e s) */ *
FROM enrollment e, student s
WHERE e.student_id = s.student_id
```

FIGURE 18.9 Nested-loop join

OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALITY
SELECT STATEMENT			229	226
NESTED LOOPS				
NESTED LOOPS			229	226
TABLE ACCESS	ENROLLMENT	FULL	3	226
INDEX	STU_PK	UNIQUE SCAN	0	1
Access Predicates				
	E.STUDENT_ID=S.ST			
TABLE ACCESS	STUDENT	BY INDEX ROWID	1	1

The execution plan for a nested loop is read differently from the other execution plans because of the loop. The access to the STU_PK index, the most indented row, is not read first but rather is probed for every row of the driving ENROLLMENT table.

The nested-loop join is typically the fastest join when the goal is to retrieve the first row as quickly as possible. It is also the best join to use when you access

819

820

approximately 1 to 10 percent of the total rows from the tables involved. This percentage varies, depending on the total number of rows returned, the total number of rows in the table, various parameters in your Oracle initialization file, and the Oracle version. It gives you a general idea of when this join is useful.



The selection of the driving table is essential to good performance of a nested-loop join. Making the driving table return the least number of rows is critical for probing fewer records in subsequent joins to other tables. Therefore, you should eliminate as many rows as possible from the driving table.

Bind Variables and the Optimizer

If you repeatedly execute the same statement, with only slightly different values in the WHERE clause, you can eliminate the parsing with the use of *bind variables*, also referred to as *host variables*. For example, if the users of your program repeatedly issue this query but substitute a different phone number each time, you should consider substituting a bind variable for the literal value.

```
SELECT last_name, first_name
FROM student
WHERE phone = '614-555-5555'
```

The use of a bind variable eliminates the parsing of the SQL statement. This overhead is significant when you have many users on a system. Bind variables are prefixed with a colon. The new statement with a bind variable looks as follows.

```
SELECT last_name, first_name
FROM student
WHERE phone = :phone_no
```

The :PHONE_NO bind variable gets a new value assigned whenever the user issues the statement. The statement with the bind variable is already parsed, and the execution plan is determined and therefore ready for execution.

The following example illustrates the use of a bind variable in SQL*Plus. The VARIABLE command creates a bind variable. The next PL/SQL block assigns the variable the value 914-555-5555.

```
SQL> VARIABLE phone_no VARCHAR2(20)
SQL> BEGIN
  2  :phone_no := '914-555-5555';
```

```
3 END;
4 .
SQL> /
```

PL/SQL procedure successfully completed.

820

821

The subsequent execution of the SQL statement using the bind variable associates the assigned value with the variable and returns the correct row.

```
SQL> SELECT last_name, first_name
2 FROM student
3 WHERE phone = :phone_no
4 /
```

```
SQL> SELECT last_name, first_name
2 FROM student
3 WHERE phone = :phone_no
4 /
```

LAST_NAME	FIRST_NAME
Mwangi	Paula

1 row selected.

Bind variables are advantageous in applications where the same statement is executed repeatedly. At the first invocation of the statement with the bind variables, the optimizer looks at the selectivity of the value and determines the best possible execution plan. This

assumes that every value associated with the bind variable has the same selectivity.

Starting with Oracle 11g, the database observes and checks whether the different bind values passed can alter the execution plan, and it may modify the execution plan accordingly. This new feature referred to as *adaptive cursor sharing*, means that Oracle may perform a modification of the execution plan, depending on the value.

In general, you will find frequent use of bind variables if the same statements are executed repeatedly. This is typically the case in transaction processing–oriented environments; the use of bind variable results in time and resource savings due to the elimination of the parsing step.

In contrast, in data warehousing environments, the queries are long running or of an ad hoc nature. Therefore, the use of literals is typically preferred because the optimizer can make an accurate determination about the most efficient execution plan. Also, the parsing time is negligible in comparison to the execution time.

For table columns where the data is not uniformly distributed, Oracle automatically creates histograms. For

example, Oracle may determine that the values of a particular column are distributed as 10 percent for values of ≥ 500 and 90 percent for values of < 500 . The histogram for this skewed data allows the optimizer to better understand the distribution of data.

Tips for Improving SQL Performance

There are several good habits to adopt when you write SQL statements. Keep the following guidelines in mind when diagnosing performance problems and when writing or rewriting SQL statements.

- Functions applied to indexed columns in the WHERE clause do not take advantage of the index unless you use the MIN or MAX function. For example, a function-based index based on the UPPER function, is useful for case-independent searches on the search column. If you always apply the TRUNC function to columns of the DATE data type, consider not storing the time portion in the column as this information is probably irrelevant. Implicit data type conversions can also cause Oracle to cast a value to another data type, thus preventing the use of the index, as you will see in the exercises.

821

822

- ▶ Use analytical functions, where possible, instead of multiple SQL statements as doing so simplifies the query writing and requires only a single pass through the table.
- ▶ Build indexes on columns you frequently use in the WHERE clause but keep in mind the performance trade-offs of DML statements. Adding an index can also adversely affect the performance of other SQL statements accessing the table. Be sure to test your scenarios carefully.
- ▶ Drop indexes that are never or very infrequently used. If you find that a specific index is used only for month-end processing, it may be advantageous to drop the index and rebuild it only shortly before the month-end job. You can also consider marking certain indexes invisible, so they are not used by the optimizer, but any related DML will continue to update them.
- ▶ Consider restructuring existing indexes. You can improve the selectivity by adding additional columns to an index. Alternatively, you can change the order of columns in a composite index.

- ▶ Full-table scans may at times be more efficient than indexes if the table is relatively small in size, as Oracle can read all the data with one I/O operation.
- ▶ If you are retrieving more than 5 to 20 percent of the rows, doing a full table scan may also be more efficient than retrieving the rows from an index.
- ▶ When joining tables using a nested-loop join, make sure to choose the table that returns the fewest number of rows as the driving table. You can enforce this with the `ORDERED` hint.
- ▶ Consider replacing the `NOT IN` operator with the `NOT EXISTS` operator and eliminating as many rows as possible in the outer query of a correlated subquery.
- ▶ If you have very large tables, consider partitioning them. This is a feature found only in Oracle Enterprise Edition.
- ▶ If your queries involve aggregates and joins against large tables, you can use materialized views to pre-store results and refresh them at set intervals. Oracle has a set of advisor procedures

that help you design and evaluate the benefits of materialized views.

- ▶ Make sure you did not forget the joining criteria so as to avoid the building of a Cartesian product.
- ▶ Rebuild or coalesce indexes periodically to improve their performance. You do this with the `ALTER INDEX indexname` command and the `REBUILD` or `COALESCE` option. This is particularly useful after many `DELETE` statements have been issued.

822

823

- ▶ Make sure statistics for your tables and indexes are periodically gathered, especially after heavy DML activity. Inaccurate statistics are a common source of performance problems. Sometimes volume and distribution of your data change and your statistics may require updating.
- ▶ Review the result of the execution plan carefully to determine the cause of any performance problem; examine execution steps involving a large number of rows/cardinality or high cost. Use hints to tune the statement and make sure the hints are valid. Do not use hints that have no effect, such as a `FIRST_ROWS_n` hint on a statement with an

GROUP BY clause because the GROUP BY operation needs to be processed first before any rows are returned.

- ▶ Use the CASE expression to avoid visiting tables multiple times. For example, if you need to aggregate rows that have different WHERE conditions within the same table, you can use the CASE expression in your SELECT list to aggregate only those rows that satisfy the necessary condition.
- ▶ SQL optimization is not just useful for queries, but also for DELETE, UPDATE, and INSERT operations. Too many indexes can slow down DML as the indexed column(s) need to be deleted or updated (if the indexed column values change) or values need to be inserted. However, indexes are beneficial for updates and deletions, particularly if the WHERE clause refers to an indexed column because the row is retrieved quickly. Missing foreign key indexes are a frequently overlooked problem; foreign key indexes are needed not only because they are frequently used in queries but also because they

can cause locking issues, as discussed in [Chapter 13](#), “Indexes, Sequences, and Views.”

- Make sure you test your SQL statements carefully in a representative test environment where the data distribution, data volume, and hardware setup is similar to a production environment. Be sure to make a copy of your statistics with the `DBMS_STATS.EXPORT_SCHEMA_STATS` procedure. In case the performance degrades after analyzing, you can restore the old statistics with the `DBMS_STATS.IMPORT_SCHEMA_STATS`.

The SQL Tuning Advisor

Oracle provides the SQL Tuning Advisor to aid in the tuning process. This advisor is accessible through the Oracle Web-based Enterprise Manager and is part of Oracle’s strategy to simplify many complex database administration and tuning tasks.

The SQL Tuning Advisor offers an automated approach to tuning, and it removes some of the guesswork by recommending specific actions that will benefit an individual statement or set of statements. The SQL Tuning Advisor incorporates the previously mentioned tips and recommends suggestions to improve the

statement. You will see that having the background knowledge acquired from the beginning of this chapter helps you evaluate the recommendations of the tool and provides you with a better understanding of the Oracle functionality. The URL to log in to the Enterprise Manager is in the format https://machine_name:1158/em.

823

[Figure 18.10](#) shows the login screen that connects with the STUDENT user and the SYSDBA privilege.

824



The SYSDBA privilege is not required to perform SQL statement tuning tasks. Alternatively, you can login with SYSMAN account and the password you set up during the installation.

Your DBA needs to set up the appropriate privileges for you to use Enterprise Manager.

FIGURE 18.10 The Enterprise Manager login screen



After the successful login, you are presented with a screen similar to [Figure 18.11](#). It shows the Enterprise Manager home page, from which you can access the Advisor Central link located on the bottom of the page, below the heading Related Links.

FIGURE 18.11 Enterprise Manager home page



Select the SQL Tuning Advisor Links link and then choose the sources for SQL statements to tune. For example, you can pick the top SQL statements to review and tune.

The SQL Tuning Advisor takes one or multiple statement and returns recommendations for the statements. You can choose between a limited or comprehensive scope. The comprehensive scope includes a SQL profile, and you can specify a time limit for the task. When the task is complete, you can view the changes to the statement.

You can accept the tuning recommendation by creating a SQL profile.

A SQL profile is for an individual statement, and it consists of additional statistics. The profile is stored in the data dictionary and is used together with the regular statistics the next time the statement is invoked. The advantage of SQL profiles is that you avoid modifying the underlying statement, which is particularly valuable with packaged applications, where you do not have the ability to access and change the SQL statement.

The SQL Access Advisor

Another advisor in the Oracle Enterprise Manager is the SQL Access Advisor. It suggests schema modifications, such as adding or dropping materialized views or indexes based on hypothetical or actual workload. The sources of the workload can be collected through Enterprise Manager. Both DBAs and application developers will find this advisor very helpful because it offers additional suggestions to further optimize an individual application.

LAB 18.1 EXERCISES

- a) Describe the result of the following query.


```
SELECT index_name, column_name,  
       column_position  
FROM user_ind_columns  
WHERE table_name = 'STUDENT'  
ORDER BY 1, 3
```

- a)** Generate an explain plan for the following SQL statement. What do you observe about the use of the index?

```
SELECT *  
FROM student  
WHERE student_id <> 101
```

- a)** Create an index called STU_FIRST_I on the FIRST_NAME column of the STUDENT table. Then execute the following SQL statement and describe the result of the execution plan.

```
SELECT student_id, first_name  
FROM student  
WHERE first_name IS NULL
```

- a)** Execute the following SQL query and describe the result of the execution plan.

```
SELECT student_id, first_name  
FROM student  
WHERE UPPER(first_name) = 'MARY'
```

825

- a) Examine the following SQL queries and their respective execution plans. What do you notice about the use of the index? Drop the index STU_FIRST_I when you are finished.

```
SELECT student_id, first_name
FROM student
WHERE first_name LIKE '%oh%'
```

Id	Operation	Name
0	SELECT STATEMENT	
1	TABLE ACCESS FULL	STUDENT

```
SELECT student_id, first_name
FROM student
WHERE first_name LIKE 'Joh%'
```

Id	Operation	Name
0	SELECT STATEMENT	
1	TABLE ACCESS BY INDEX ROWID	STUDENT
2	INDEX RANGE SCAN	STU_FIRST_I

- a) Execute the following SQL query and describe the result of the execution plan.

```
SELECT *
FROM zipcode
```

```
WHERE zip = 10025
```

- a) Explain why the following query does not use an index.

```
SELECT *  
FROM grade  
WHERE grade_type_code = 'HW'
```

- a) Given the following SELECT statement and the resulting execution plan, determine the driving table and the type of join performed.

```
SELECT --+ FIRST_ROWS(10)  
       i.last_name, c.description,  
       c.course_no  
FROM   course c, section s,  
       instructor i  
WHERE  c.course_no = s.course_no  
       AND s.instructor_id =  
       i.instructor_id  
       AND s.section_id = 133
```

Id	Operation	Name
0	SELECT STATEMENT	
1	NESTED LOOPS	
2	NESTED LOOPS	
3	TABLE ACCESS BY INDEX ROWID	SECTION
4	INDEX UNIQUE SCAN	SECT_PK
5	TABLE ACCESS BY INDEX ROWID	COURSE
6	INDEX UNIQUE SCAN	CRSE_PK
7	TABLE ACCESS BY INDEX ROWID	INSTRUCTOR
8	INDEX UNIQUE SCAN	INST_PK

- a)** The following SQL statements result in different execution plans. What differences, do you observe?

```
SELECT *
  FROM student
 WHERE student_id NOT IN
        (SELECT student_id
         FROM enrollment)
```

```
SELECT *
  FROM student s
 WHERE NOT EXISTS
        (SELECT 'X'
         FROM enrollment
         WHERE s.student_id =
student_id)
```

```
SELECT student_id
  FROM student
MINUS
SELECT student_id
  FROM enrollment
```

- a)** Show the execution plans for the following SELECT statements and describe the difference.

```
SELECT student_id, last_name,
'student'
```

```
FROM student
UNION
SELECT instructor_id, last_name,
'instructor'
FROM instructor

SELECT student_id, last_name,
'student'
FROM student
UNION ALL
SELECT instructor_id, last_name,
'instructor'
FROM instructor
```

827

828

LAB 18.1 EXERCISE ANSWERS

- a) Describe the result of the following query.

```
SELECT index_name, column_name,
column_position
FROM user_ind_columns
WHERE table_name = 'STUDENT'
ORDER BY 1, 3
```

ANSWER: The result of the query shows a listing of all indexes on the STUDENT table and the order in which the columns are indexed. In this example, both indexes are single-column indexes.

INDEX_NAME	COLUMN_NAME	COLUMN_POSITION
STU_ZIP_FK_I	ZIP	1
STU_PK	STUDENT_ID	1

2 rows selected.

- a) Generate an explain plan for the following SQL statement. What do you observe about the use of the index?

```
SELECT *
FROM student
WHERE student_id <> 101
```

ANSWER: The index is not used in this query; every record is examined with the full table scan instead.

Id	Operation	Name
0	SELECT STATEMENT	
1	TABLE ACCESS FULL	STUDENT

Inequality conditions, such as \neq , \neq , or any negation using NOT, typically do not make use of an index.

- a) Create an index called STU_FIRST_I on the FIRST_NAME column of the STUDENT table.

Then execute the following SQL statement and describe the result of the execution plan.

```
SELECT student_id, first_name
FROM student
WHERE first_name IS NULL
```

ANSWER: The query does not make use of the index on the FIRST_NAME column because NULL values are not stored in the index. Therefore, a full table scan is executed.

```
CREATE INDEX stu_first_i ON
student(first_name)
```

Index created.

828

The subsequently issued query results in the following execution plan.

Id	Operation	Name	

0	SELECT STATEMENT		
1	TABLE ACCESS FULL	STUDENT	

829

If you expect to execute this query frequently and want to avoid a full table scan, you might want to consider adding a row with a default value for FIRST_NAME, such as 'Unknown'. When this value is inserted in the index, a subsequently

issued query, such as the following, uses the index.

```
SELECT student_id, first_name
FROM student
WHERE first_name = 'Unknown'
```

The index is not efficient, however, if you expect a significantly large number of the values to be 'Unknown'. In this case, retrieving values through the index rather than the full table scan may take longer.

- a) Execute the following SQL query and describe the result of the execution plan.

```
SELECT student_id, first_name
FROM student
WHERE UPPER(first_name) = 'MARY'
```

ANSWER: The query does not make use of the index on the FIRST_NAME column.

Id	Operation	Name

0	SELECT STATEMENT	
1	TABLE ACCESS FULL	STUDENT

You can use the UPPER function in the SQL statement if you are unsure in which case the first name was entered.

The query returns records with the values MARY, Mary, or combinations thereof. Each time you modify an indexed column, the use of the index is disabled. The solution is to create a function-based index. For more information on this topic, refer to [Chapter 13](#).

- a) Examine the following SQL queries and their respective execution plans. What do you notice about the use of the index? Drop the index STU_FIRST_I.

```
SELECT student_id, first_name
FROM student
WHERE first_name LIKE '%oh%'
```

Id	Operation	Name	

0	SELECT STATEMENT		
1	TABLE ACCESS FULL	STUDENT	

829

```
SELECT student_id, first_name
FROM student
WHERE first_name LIKE 'Joh%'
```

830

Id	Operation	Name
0	SELECT STATEMENT	
1	TABLE ACCESS BY INDEX ROWID	STUDENT
2	INDEX RANGE SCAN	STU_FIRST_I

ANSWER: The first query does not make use of the index on the FIRST_NAME column because the index cannot determine the index entries. The second query allows the use of the index.

You may find that the second query does not use an index. The optimizer may determine that the table is so small that it is more efficient to read the entire table. However, if the table contains a larger data set, the optimizer will make use of the index.

You may also see an execution plan that takes advantage of both the STU_FIRST_I index and the STU_PK primary key index. If the columns in the query can be satisfied by using both indexes, you may not see any table access. For example, you might get an execution plan that retrieves the ROWIDs based on the WHERE clause. These ROWIDS are then joined to

ROWIDs of the primary key index to obtain the STUDENT_ID column values.

Drop the index from the schema to restore the schema to its original state.

```
DROP INDEX stu_first_i
```

Index dropped.

- a) Execute the following SQL query and describe the result of the execution plan.

```
SELECT *  
FROM zipcode  
WHERE zip = 10025
```

ANSWER: The query does not make use of the primary key index on the ZIP column.

Id	Operation	Name	

0	SELECT STATEMENT		
1	TABLE ACCESS FULL	ZIPCODE	

The full table access is used because the data types between the ZIP column and the number literal do not agree. The ZIP column is of VARCHAR2 data type to store leading zeros for zip codes such as 00706, and the literal in the

WHERE clause is a NUMBER data type. The following query is an example of Oracle performing an implicit conversion. Oracle converts the ZIP column to a NUMBER data type and, therefore, disables the use of the index. If the WHERE clause is written as follows, it uses the index.

830

WHERE zip = '10025'

831

Id	Operation	Name
0	SELECT STATEMENT	
1	TABLE ACCESS BY INDEX ROWID	ZIPCODE
2	INDEX UNIQUE SCAN	ZIP_PK

- a) Explain why the following query does not use an index.

```
SELECT *
FROM grade
WHERE grade_type_code = 'HW'
```

ANSWER: The GRADE_TYPE_CODE column is not the leading column on any index of the GRADE table.

Id	Operation	Name
0	SELECT STATEMENT	
1	TABLE ACCESS FULL	GRADE

The following query shows the indexes on the GRADE table. The GRADE_TYPE_CODE is a column in two different indexes but is never the leading column, nor are any of the leading columns in the WHERE clause of the query.

```
SELECT index_name, column_name,
column_position
FROM user_ind_columns
WHERE table_name = 'GRADE'
ORDER BY 1, 3
```

```
SELECT index_name, column_name, column_position
FROM user_ind_columns
WHERE table_name = 'GRADE'
ORDER BY 1, 3
```

INDEX_NAME	COLUMN_NAME	COLUMN_POSITION
GR_GRTW_FK_I	SECTION_ID	1
GR_GRTW_FK_I	GRADE_TYPE_CODE	2
GR_PK	STUDENT_ID	1
GR_PK	SECTION_ID	2
GR_PK	GRADE_TYPE_CODE	3
GR_PK	GRADE_CODE_OCCURRENCE	4

```
6 rows selected.
```

The following query makes use of the index GR_GRTW_FK_I because the leading edge of the index is in the WHERE clause.

```
SELECT *
FROM grade
```

```
WHERE grade_type_code = 'HW'
AND section_id = 123
```

831

And the following query uses the primary key index GR_PK.

832

```
SELECT *
FROM grade
WHERE grade_type_code = 'HW'
AND section_id = 123
AND student_id = 567
```

Oracle's skip scan feature improves index scans when the leading portion of the index is not specified. Essentially, scanning an index is faster than scanning the table, and the skip scanning feature splits the index into smaller sub-indexes. These different sub-indexes show the number of distinct values in the leading index. The feature is most useful when there are few distinct values in the leading column of the index. The explain plan indicates whether Oracle took advantage of the skip scan feature to access the data.

- a) Given the following SELECT statement and the resulting execution plan, determine the driving table and the type of join performed.

```
SELECT --+ FIRST_ROWS(10)
       i.last_name, c.description, c.course_no
FROM   course c, section s, instructor i
WHERE  c.course_no = s.course_no
       AND s.instructor_id = i.instructor_id
       AND s.section_id = 133
```

Id	Operation	Name
0	SELECT STATEMENT	
1	NESTED LOOPS	
2	NESTED LOOPS	
3	TABLE ACCESS BY INDEX ROWID	SECTION
4	INDEX UNIQUE SCAN	SECT_PK
5	TABLE ACCESS BY INDEX ROWID	INSTRUCTOR
6	INDEX UNIQUE SCAN	INST_PK
7	TABLE ACCESS BY INDEX ROWID	COURSE
8	INDEX UNIQUE SCAN	CRSE_PK

ANSWER: The driving table of this nested-loop join is the SECTION table.

The query performs the following steps: The index SECT_PK is probed for the SECTION_ID 133, and one record in the SECTION table is accessed. Then the index INST_PK on the INSTRUCTOR table is checked for the matching INSTRUCTOR_ID. The LAST_NAME column value is retrieved from INSTRUCTOR. Finally, the COURSE index CRSE_PK is used to find a match for the COURSE_NO values, based on the initial result set of the SECTION table and the

desired DESCRIPTION column from the COURSE table retrieved.

You can influence the join order with the ORDERED hint or the LEADING hint. This can have a significant impact on performance, depending on the size and WHERE conditions of the join. Try to eliminate as many rows as possible from the driving table to avoid probing many subsequent inner tables.

Comparing the execution time of a nested-loop join to a sort-merge join or a hash join probably will not show a great variance for the data within the STUDENT schema, but if you are joining larger tables, the differences may be significant.

832

- a) The following SQL statements result in different execution plans. What differences do you observe?

```
SELECT *  
FROM student  
WHERE student_id NOT IN  
  (SELECT student_id  
   FROM enrollment)
```

```
SELECT *  
FROM student s
```

833


```
WHERE NOT EXISTS
(SELECT 'X'
FROM enrollment
WHERE s.student_id = student_id)

SELECT student_id
FROM student
MINUS
SELECT student_id
FROM enrollment
```

ANSWER: Each query has a different explain plan.

The NOT IN subquery does not take advantage of the index on the ENROLLMENT table, resulting in a full table scan.

Id		Operation	Name

0		SELECT STATEMENT	
1		FILTER	
2		TABLE ACCESS FULL	STUDENT
3		TABLE ACCESS FULL	ENROLLMENT

The NOT IN operator can be very inefficient. Consider replacing it with NOT EXISTS, particularly because the NOT EXISTS operator takes advantage of the index on the

ENROLLMENT table (see the following explain plan). You should keep in mind that you want to eliminate as many rows as possible in the outer query of the NOT EXISTS correlated subquery in order to minimize the repeated execution of the inner query.

Id	Operation	Name
0	SELECT STATEMENT	
1	FILTER	
2	TABLE ACCESS FULL	STUDENT
3	INDEX RANGE SCAN	ENR_PK

833

834



Depending on the Oracle version, you may get different results from the listed explain plans in this book. With each new Oracle version, modifications are made to the optimizer that frequently result in more efficient execution plans.

The execution plan of the MINUS operator does not look very impressive, but using it can actually be one of the fastest ways to retrieve the

result, especially when a large number of records are involved.

Id	Operation	Name
0	SELECT STATEMENT	
1	MINUS	
2	SORT UNIQUE NOSORT	
3	INDEX FULL SCAN	STU_PK
4	SORT UNIQUE NOSORT	
5	INDEX FULL SCAN	ENR_PK



Always consider alternative SQL syntax when writing queries and tune your SQL statements with a representative data set. If the distribution of the data changes, so will the statistics, and the optimizer may favor a different execution plan.

- a) Show the execution plans for the following SELECT statements and describe the difference.

```
SELECT student_id, last_name,  
       'student'  
FROM student  
UNION
```

```
SELECT instructor_id, last_name,
'instructor'
FROM instructor
SELECT student_id, last_name,
'student'
FROM student
UNION ALL
SELECT instructor_id, last_name,
'instructor'
FROM instructor
```

ANSWER: The UNION statement involves an additional sort that is not performed on the UNION ALL statement.

```
SELECT student_id, last_name,
'student'
FROM student
UNION
SELECT instructor_id, last_name,
'instructor'
FROM instructor
```

834

835

Id	Operation	Name
0	SELECT STATEMENT	
1	SORT UNIQUE	
2	UNION	
3	TABLE ACCESS FULL	STUDENT
4	TABLE ACCESS FULL	INSTRUCTOR

```
SELECT student_id, last_name,  
       'student'  
FROM student  
UNION ALL  
SELECT instructor_id, last_name,  
       'instructor'  
FROM instructor
```

Id		Operation	Name

0		SELECT STATEMENT	
1		UNION-ALL	
2		TABLE ACCESS FULL	STUDENT
3		TABLE ACCESS FULL	INSTRUCTOR

Whenever possible, avoid any unnecessary sorts required by the use of UNION or DISTINCT.

835

836

Lab 18.1 Quiz

In order to test your progress, you should be able to answer the following questions.

1) The optimizer chooses from among many possible execution plans the one with the lowest cost.

_____ a) True

_____ b) False

2) An ORDERED hint can influence the join order of SQL statements using the CBO.

_____ **a)** True

_____ **b)** False

3) The join order of tables is irrelevant to the performance of the nested-loop join.

_____ **a)** True

_____ **b)** False

4) Incorrectly written hints are treated as comments and ignored.

_____ **a)** True

_____ **b)** False

5) The gathered statistics are stored in the data dictionary.

_____ **a)** True

_____ **b)** False

ANSWERS APPEAR IN APPENDIX A.

836

WORKSHOP

The projects in this section are meant to prompt you to utilize all the skills you have acquired throughout this chapter. The answers to these projects can be found at the companion Web site to this book, located at www.oraclesqlbyexample.com.

- 1) Given the following execution plan, describe the steps and their order of execution.

```
SELECT c.course_no,  
c.description,  
i.instructor_id  
FROM course c, section s,  
instructor i  
WHERE prerequisite = 30  
AND c.course_no = s.course_no  
AND s.instructor_id =  
i.instructor_id
```

Id	Operation	Name
0	SELECT STATEMENT	
1	NESTED LOOPS	
2	NESTED LOOPS	
3	TABLE ACCESS BY INDEX ROWID	COURSE
4	INDEX RANGE SCAN	CRSE_CRSE_FK_I
5	TABLE ACCESS BY INDEX ROWID	SECTION
6	INDEX RANGE SCAN	SECT_CRSE_FK_I
7	INDEX UNIQUE SCAN	INST_PK

2) Describe the steps of the following execution plan.

```
UPDATE enrollment e
SET final_grade =
(SELECT NVL(AVG(numeric_grade), 0)
FROM grade
WHERE e.student_id = student_id
AND e.section_id = section_id)
WHERE student_id = 1000
AND section_id = 2000
0 rows updated.
```

Id	Operation	Name
0	UPDATE STATEMENT	
1	UPDATE	ENROLLMENT
2	INDEX UNIQUE SCAN	ENR_PK
3	SORT AGGREGATE	
4	TABLE ACCESS BY INDEX ROWID	GRADE
5	INDEX RANGE SCAN	GR_PK

837

3) The following SQL statement has an error in the hint. Correct the statement so Oracle can use the hint.

```
SELECT /*+ INDEX (student stu_pk)
*/ *
FROM student s
```

838


```
WHERE last_name = 'Smith'
```