**CHAPTER 16  
SQL Tuning**

In this chapter, you will do the following:

Image Learn about SQL tuning

Image See SQL tuning tips that you can use to shorten the length of time your queries take to execute

Image Learn about the Oracle optimizer

Image See how to compare the cost of performing queries

Image Examine optimizer hints

Image Learn about some additional tuning tools

**INTRODUCING SQL TUNING**

One of the main strengths of SQL is that you don’t have to tell the database exactly how to obtain the data requested. You simply run a query specifying the information you want, and the database software figures out the best way to get it. Sometimes, you can improve the performance of your SQL statements by "tuning" them. In the following sections, you’ll see tuning tips that can make your queries run faster; later, you’ll see more advanced tuning techniques.

**USE A WHERE CLAUSE TO FILTER ROWS**

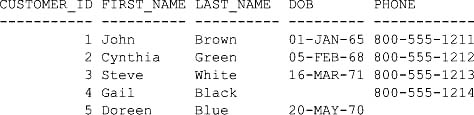
Many novices retrieve all the rows from a table when they only want one row (or a few rows). This is very wasteful. A better approach is to add a WHERE clause to a query. That way, you restrict the rows retrieved to just those actually needed.

For example, say you want the details for customer #1 and #2. The following query retrieves all the rows from the customers table in the store schema (wasteful):

-- BAD (retrieves all rows from the customers table)

**SELECT \***

**FROM customers;**



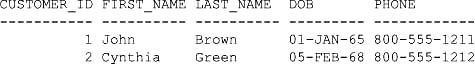
The next query adds a WHERE clause to the previous example to just get customer #1 and #2:

-- GOOD (uses a WHERE clause to limit the rows retrieved)

**SELECT \***

**FROM customers**

**WHERE customer\_id IN (1, 2);**



You should avoid using functions in the WHERE clause, as that increases execution time.

**USE TABLE JOINS RATHER THAN MULTIPLE QUERIES**

If you need information from multiple related tables, you should use join conditions rather than multiple queries. In the following bad example, two queries are used to get the product name and the product type name for product #1 (using two queries is wasteful). The first query gets the name and product\_type\_id column values from the products table for product #1. The second query then uses that product\_type\_id to get the name column from the product\_types table.

-- BAD (two separate queries when one would work)

**SELECT name, product\_type\_id**

**FROM products**

**WHERE product\_id = 1;**

NAME PRODUCT\_TYPE\_ID

------------------------------ ---------------

Modern Science 1

**SELECT name**

**FROM product\_types**

**WHERE product\_type\_id = 1;**

NAME

----------

Book

Instead of using the two queries, you should write one query that uses a join between the products and product\_types tables. The following good query shows this:

-- GOOD (one query with a join)

**SELECT p.name, pt.name**

**FROM products p, product\_types pt**

**WHERE p.product\_type\_id = pt.product\_type\_id**

**AND p.product\_id = 1;**

NAME NAME

------------------------------ ----------

Modern Science Book

This query results in the same product name and product type name being retrieved as in the first example, but the results are obtained using one query. One query is generally more efficient than two.

You should choose the join order in your query so that you join fewer rows to tables later in the join order. For example, say you were joining three related tables named tab1, tab2, and tab3. Assume tab1 contains 1,000 rows, tab2 100 rows, and tab3 10 rows. You should join tab1 with tab2 first, followed by tab2 and tab3.

Also, avoid joining complex views in your queries, because doing so causes the queries for the views to be run first, followed by your actual query. Instead, write your query using the tables rather than the views.

**USE FULLY QUALIFIED COLUMN REFERENCES WHEN PERFORMING JOINS**

Always include table aliases in your queries and use the alias for each column in your query (this is known as "fully qualifying" your column references). That way, the database doesn’t have to search for each column in the tables used in your query.

The following bad example uses the aliases p and pt for the products and product\_types tables, respectively, but the query doesn’t fully qualify the description and price columns:

-- BAD (description and price columns not fully qualified)

**SELECT p.name, pt.name, description, price**

**FROM products p, product\_types pt**

**WHERE p.product\_type\_id = pt.product\_type\_id**

**AND p.product\_id = 1;**

NAME NAME

------------------------------ ----------

DESCRIPTION PRICE

-------------------------------------------------- ----------

Modern Science Book

A description of modern science 19.95

This example works, but the database has to search both the products and product\_types tables for the description and price columns; that’s because there’s no alias that tells the database which table those columns are in. The extra time spent by the database having to do the search is wasted time.

The following good example includes the table alias p to fully qualify the description and price columns:

-- GOOD (all columns are fully qualified)

**SELECT p.name, pt.name, p.description, p.price**

**FROM products p, product\_types pt**

**WHERE p.product\_type\_id = pt.product\_type\_id**

**AND p.product\_id = 1;**

NAME NAME

------------------------------ ----------

DESCRIPTION PRICE

-------------------------------------------------- ----------

Modern Science Book

A description of modern science 19.95

Because all references to columns include a table alias, the database doesn’t have to waste time searching the tables for the columns, and execution time is reduced.

**USE CASE EXPRESSIONS RATHER THAN MULTIPLE QUERIES**

Use CASE expressions rather than multiple queries when you need to perform many calculations on the same rows in a table. The following bad example uses multiple queries to count the number of products within various price ranges:

-- BAD (three separate queries when one CASE statement would work)

**SELECT COUNT(\*)**

**FROM products**

**WHERE price < 13;**

COUNT(\*)

----------

2

**SELECT COUNT(\*)**

**FROM products**

**WHERE price BETWEEN 13 AND 15;**

COUNT(\*)

----------

5

**SELECT COUNT(\*)**

**FROM products**

**WHERE price > 15;**

COUNT(\*)

----------

5

Rather than using three queries, you should write one query that uses CASE expressions. This is shown in the following good example:

-- GOOD (one query with a CASE expression)

**SELECT**

**COUNT(CASE WHEN price < 13 THEN 1 ELSE null END) low**,

**COUNT(CASE WHEN price BETWEEN 13 AND 15 THEN 1 ELSE null END) med**,

**COUNT(CASE WHEN price > 15 THEN 1 ELSE null END) high**

**FROM products;**

LOW MED HIGH

---------- ---------- ----------

2 5 5

Notice that the counts of the products with prices less than $13 are labeled as low, products between $13 and $15 are labeled med, and products greater than $15 are labeled high.

Image

**NOTE**  
*You can, of course, use overlapping ranges and different functions in your* CASE *expressions*.

**ADD INDEXES TO TABLES**

When looking for a particular topic in a book, you can either scan the whole book or use the index to find the location. An index for a database table is similar in concept to a book index, except that database indexes are used to find specific rows in a table. The downside of indexes is that when a row is added to the table, additional time is required to update the index for the new row.

Generally, you should create an index on a column when you are retrieving a small number of rows from a table containing many rows. A good rule of thumb is

*Create an index when a query retrieves <= 10 percent of the total rows in a table*.

This means the column for the index should contain a wide range of values. A good candidate for indexing would be a column containing a unique value for each row (for example, a social security number). A poor candidate for indexing would be a column that contains only a small range of values (for example, N, S, E, W or 1, 2, 3, 4, 5, 6). An Oracle database automatically creates an index for the primary key of a table and for columns included in a unique constraint.

In addition, if your database is accessed using a lot of hierarchical queries (that is, a query containing a CONNECT BY), you should add indexes to the columns referenced in the START WITH and CONNECT BYclauses (see [Chapter 7](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch07.html#ch07) for details on hierarchical queries).

Finally, for a column that contains a small range of values and is frequently used in the WHERE clause of queries, you should consider adding a bitmap index to that column. Bitmap indexes are typically used in data warehouses, which are databases containing very large amounts of data. The data in a data warehouse is typically read using many queries, but the data is not modified by many concurrent transactions.

Normally, a database administrator is responsible for creating indexes. However, as an application developer, you’ll be able to provide the DBA with feedback on which columns are good candidates for indexing, because you may know more about the application than the DBA. [Chapter 10](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch10.html#ch10) covers indexes in depth, and you should re-read the section on indexes if necessary.

**USE WHERE RATHER THAN HAVING**

You use the WHERE clause to filter rows; you use the HAVING clause to filter groups of rows. Because the HAVING clause filters groups of rows *after* they have been grouped together (which takes some time to do), you should first filter rows using a WHERE clause whenever possible. That way, you avoid the time taken to group the filtered rows together in the first place.

The following bad query retrieves the product\_type\_id and average price for products whose product\_type\_id is 1 or 2. To do this, the query performs the following:

Image It uses the GROUP BY clause to group rows into blocks with the same product\_type\_id.

Image It uses the HAVING clause to filter the returned results to those groups that have a product\_type\_id in 1 or 2 (this is bad, because a WHERE clause would work).

-- BAD (uses HAVING rather than WHERE)

**SELECT product\_type\_id, AVG(price)**

**FROM products**

**GROUP BY product\_type\_id**

**HAVING product\_type\_id IN (1, 2);**

PRODUCT\_TYPE\_ID AVG(PRICE)

--------------- ----------

1 24.975

2 26.22

The following good query rewrites the previous example to use WHERE rather than HAVING to first filter the rows to those whose product\_type\_id is 1 or 2:

-- GOOD (uses WHERE rather than HAVING)

**SELECT product\_type\_id, AVG(price)**

**FROM products**

**WHERE product\_type\_id IN (1, 2)**

**GROUP BY product\_type\_id;**

PRODUCT\_TYPE\_ID AVG(PRICE)

--------------- ----------

1 24.975

2 26.22

**USE UNION ALL RATHER THAN UNION**

You use UNION ALL to get all the rows retrieved by two queries, including duplicate rows; you use UNION to get all non-duplicate rows retrieved by the queries. Because UNION removes duplicate rows (which takes some time to do), you should use UNION ALL whenever possible.

The following bad query uses UNION (bad because UNION ALL would work) to get the rows from the products and more\_products tables; notice that all non-duplicate rows from products and more\_products are retrieved:

-- BAD (uses UNION rather than UNION ALL)

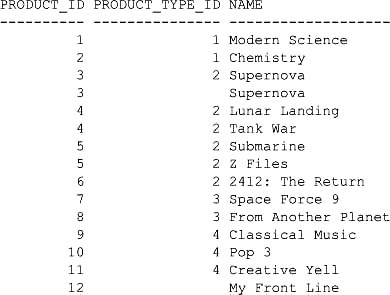
**SELECT product\_id, product\_type\_id, name**

**FROM products**

**UNION**

**SELECT prd\_id, prd\_type\_id, name**

**FROM more\_products;**



The following good query rewrites the previous example to use UNION ALL; notice that all the rows from products and more\_products are retrieved, including duplicates:

-- GOOD (uses UNION ALL rather than UNION)

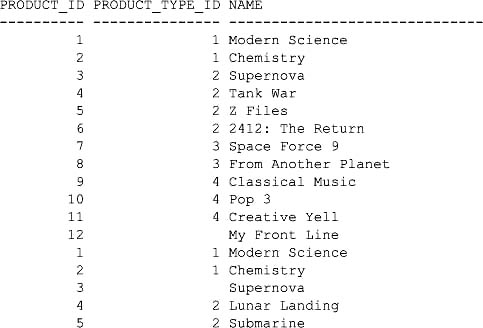
**SELECT product\_id, product\_type\_id, name**

**FROM products**

**UNION ALL**

**SELECT prd\_id, prd\_type\_id, name**

**FROM more\_products;**



**USE EXISTS RATHER THAN IN**

You use IN to check if a value is contained in a list. You use EXISTS to check for the existence of rows returned by a subquery. EXISTS is different from IN: EXISTS just checks for the existence of rows, whereas IN checks actual values. EXISTS typically offers better performance than IN with subqueries. Therefore, you should use EXISTS rather than IN whenever possible.

You should refer back to the section entitled "Using EXISTS and NOT EXISTS with a Correlated Subquery" in [Chapter 6](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch06.html#ch06) for full details on when you should use EXISTS with a correlated subquery (an important point to remember is that correlated subqueries can resolve null values).

The following bad query uses IN (bad because EXISTS would work) to retrieve products that have been purchased:

-- BAD (uses IN rather than EXISTS)

**SELECT product\_id, name**

**FROM products**

**WHERE product\_id IN**

**(SELECT product\_id**

**FROM purchases);**

PRODUCT\_ID NAME

---------- -----------------------------

1 Modern Science

2 Chemistry

3 Supernova

The following good query rewrites the previous example to use EXISTS:

-- GOOD (u ses EXISTS rather than IN)

**SELECT product\_id, name**

**FROM products outer**

**WHERE EXISTS**

**(SELECT 1**

**FROM purchases inner**

**WHERE inner.product\_id = outer.product\_id);**

PRODUCT\_ID NAME

---------- -----------------------------

1 Modern Science

2 Chemistry

3 Supernova

**USE EXISTS RATHER THAN DISTINCT**

You can suppress the display of duplicate rows using DISTINCT. You use EXISTS to check for the existence of rows returned by a subquery. Whenever possible, you should use EXISTS rather than DISTINCT, because DISTINCT sorts the retrieved rows before suppressing the duplicate rows.

The following bad query uses DISTINCT (bad because EXISTS would work) to retrieve products that have been purchased:

-- BAD (uses DISTINCT when EXISTS would work)

**SELECT DISTINCT pr.product\_id, pr.name**

**FROM products pr, purchases pu**

**WHERE pr.product\_id = pu.product\_id;**

PRODUCT\_ID NAME

---------- -----------------------------

1 Modern Science

2 Chemistry

3 Supernova

The following good query rewrites the previous example to use EXISTS rather than DISTINCT:

-- GOOD (uses EXISTS rather than DISTINCT)

**SELECT product\_id, name**

**FROM products outer**

**WHERE EXISTS**

**(SELECT 1**

**FROM purchases inner**

**WHERE inner.product\_id = outer.product\_id);**

PRODUCT\_ID NAME

---------- -----------------------------

1 Modern Science

2 Chemistry

3 Supernova

**USE GROUPING SETS RATHER THAN CUBE**

The GROUPING SETS clause typically offers better performance than CUBE. Therefore, you should use GROUPING SETS rather than CUBE wherever possible. This is fully covered in the section entitled "Using the GROUPING SETS Clause" in [Chapter 7](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch07.html#ch07).

**USE BIND VARIABLES**

The Oracle database software caches SQL statements; a cached SQL statement is reused if an identical statement is submitted to the database. When an SQL statement is reused, the execution time is reduced. However, the SQL statement must be *absolutely identical* in order for it to be reused. This means that

Image All characters in the SQL statement must be the same.

Image All letters in the SQL statement must be in the same case.

Image All spaces in the SQL statement must be the same.

If you need to supply different column values in a statement, you can use bind variables instead of literal column values. You’ll see examples that clarify these ideas next.

**Non-Identical SQL Statements**

In this section, you’ll see some non-identical SQL statements. The following non-identical queries retrieve products #1 and #2:

SELECT \* FROM products WHERE product\_id = 1;

SELECT \* FROM products WHERE product\_id = 2;

These queries are not identical, because the value 1 is used in the first statement, but the value 2 is used in the second.

The following non-identical queries have spaces in different positions:

SELECT \* FROM products WHERE product\_id = 1;

SELECT \* FROM products WHERE product\_id = 1;

The following non-identical queries use a different case for some of the characters:

select \* from products where product\_id = 1;

SELECT \* FROM products WHERE product\_id = 1;

Now that you’ve seen some non-identical statements, let’s take a look at identical SQL statements that use bind variables.

**Identical SQL Statements That Use Bind Variables**

You can ensure that a statement is identical by using bind variables to represent column values. You create a bind variable using the SQL\*Plus VARIABLE command. For example, the following command creates a variable named v\_product\_id of type NUMBER:

**VARIABLE v\_product\_id NUMBER**

Image

**NOTE**  
*You can use the types shown in*[*Table A-1*](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/app.html#table_A-1)*of the appendix to define the type of a bind variable*.

You reference a bind variable in an SQL or PL/SQL statement using a colon followed by the variable name (such as : v\_product\_id). For example, the following PL/SQL block sets v\_product\_id to 1:

**BEGIN**

**:v\_product\_id := 1;**

**END;**

**/**

The following query uses v\_product\_id to set the product\_id column value in the WHERE clause; because v\_product\_id was set to 1 in the previous PL/SQL block, the query retrieves the details of product #1:

**SELECT \* FROM products WHERE product\_id = :v\_product\_id;**

PRODUCT\_ID PRODUCT\_TYPE\_ID NAME

---------- --------------- ------------------------------

DESCRIPTION PRICE

-------------------------------------------------- ----------

1 1 Modern Science

A description of modern science 19.95

The next example sets v\_product\_id to 2 and repeats the query:

**BEGIN**

**:v\_product\_id := 2;**

**END;**

**/**

**SELECT \* FROM products WHERE product\_id = :v\_product\_id;**

PRODUCT\_ID PRODUCT\_TYPE\_ID NAME

---------- --------------- ------------------------------

DESCRIPTION PRICE

-------------------------------------------------- ----------

2 1 Chemistry

Introduction to Chemistry 30

Because the query used in this example is identical to the previous query, the cached query is reused and there’s an improvement in performance.

Image

**TIP**  
*You should typically use bind variables if you’re performing the same query many times. Also, in the example, the bind variables are session specific and need to be reset if the session is lost*.

**Listing and Printing Bind Variables**

You list bind variables in SQL\*Plus using the VARIABLE command. For example:

**VARIABLE**

variable v\_product\_id

datatype NUMBER

You display the value of a bind variable in SQL\*Plus using the PRINT command. For example:

**PRINT v\_product\_id**

V\_PRODUCT\_ID

-------------

2

**Using a Bind Variable to Store a Value Returned by a PL/SQL Function**

You can also use a bind variable to store returned values from a PL/SQL function. The following example creates a bind variable named v\_average\_product\_price and stores the result returned by the function average\_product\_price() (this function was described in [Chapter 11](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch11.html#ch11) and calculates the average product price for the supplied product\_type\_id):

**VARIABLE v\_average\_product\_price NUMBER**

**BEGIN**

**:v\_average\_product\_price := average\_product\_price(1);**

**END;**

**/**

**PRINT v\_average\_product\_price**

V\_AVERAGE\_PRODUCT\_PRICE

------------------------

24.975

**Using a Bind Variable to Store Rows from a REFCURSOR**

You can also use a bind variable to store returned values from a REFCURSOR (a REFCURSOR is a pointer to a list of rows). The following example creates a bind variable named v\_products\_refcursor and stores the result returned by the function product\_package.get\_products\_ref\_cursor() (this function was introduced in [Chapter 11](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch11.html#ch11); it returns a pointer to the rows in the products table):

**VARIABLE v\_products\_refcursor REFCURSOR**

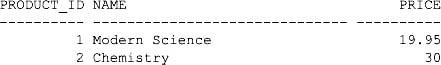
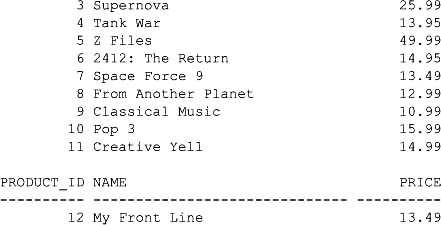
**BEGIN**

**:v\_products\_refcursor := product\_package.get\_products\_ref\_cursor();**

**END;**

**/**

**PRINT v\_products\_refcursor**

**COMPARING THE COST OF PERFORMING QUERIES**

The Oracle database software uses a subsystem known as the *optimizer* to generate the most efficient path to access the data stored in the tables. The path generated by the optimizer is known as an *execution plan*. Oracle Database 10*g* and above automatically gathers statistics about the data in your tables and indexes in order to generate the best execution plan (this is known as *cost-based* optimization).

Comparing the execution plans generated by the optimizer allows you to judge the relative cost of one SQL statement versus another. You can use the results to improve your SQL statements. In this section, you’ll learn how to view and interpret a couple of example execution plans.

Image

**NOTE**  
*Database versions prior to Oracle Database 10g don’t automatically gather statistics, and the optimizer automatically defaults to* rule-based *optimization. Rule-based optimization uses syntactic rules to generate the execution plan. Cost-based optimization is typically better than rule-based optimization because the former uses actual information gathered from the data in the tables and indexes. If you’re using Oracle Database* 9i *or below, you can gather statistics yourself (you’ll learn how to do that later in the section "Gathering Table Statistics")*.

**Examining Execution Plans**

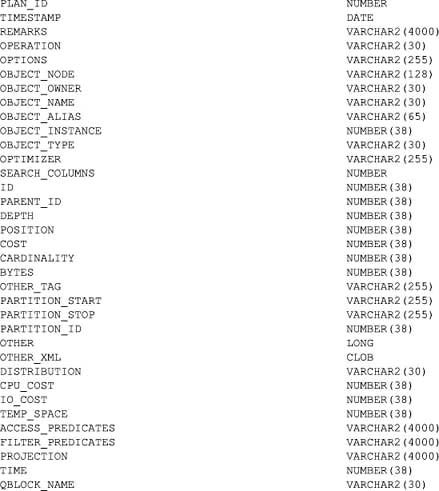
The optimizer generates an execution plan for an SQL statement. You can examine the execution plan using the SQL\*Plus EXPLAIN PLAN command. The EXPLAIN PLAN command populates a table named plan\_table with the SQL statement’s execution plan (plan\_table is often referred to as the "plan table"). You may then examine that execution plan by querying the plan table. The first thing you must do is check if the plan table currently exists in the database.

**Checking if the Plan Table Currently Exists in the Database**

To check if the plan table currently exists in the database, you should connect to the database as the store user and run the following DESCRIBE command:

SQL> **DESCRIBE plan\_table**

Image



If you get a table description similar to these results, you have the plan table already. If you get an error, then you need to create the plan table.

**Creating the Plan Table**

If you don’t have the plan table, you must create it. To do this, you run the SQL\*Plus script utlxplan.sql(on my Windows computer, the script is located in the directory E:\oracle\_11g\product\11.1.0\db\_1\RDBMS\ADMIN). The following example shows the command to run the utlxplan.sql script:

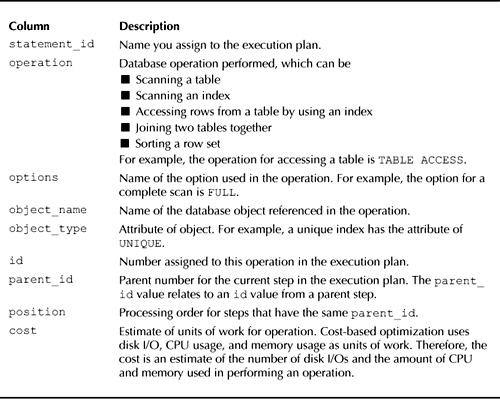
SQL> **@ E:\oracle\_11g\product\11.1.0\db\_1\RDBMS\ADMIN\utlxplan.sql**

Image

**NOTE**  
*You’ll need to replace the directory path with the path for your environment*.

The most important columns in the plan table are shown in [Table 16-1](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch16.html#table_16-1).

|  |
| --- |
| **Creating a Central Plan Table** |
| If necessary, a database administrator can create one central plan table. That way, individual users don’t have to create their own plan tables. To do this, a database administrator performs the following steps: |
| **1**. Creates the plan table in a schema of their choice by running the utlxplan.sql script |
| **2**. Creates a public synonym for the plan table |
| **3**. Grants access on the plan table to the public role |
| Here is an example of these steps: |
| **@ E:\oracle\_11g\product\11.1.0\db\_1\RDBMS\ADMIN\utlxplan.sql CREATE PUBLIC SYNONYM plan\_table FOR plan\_table; GRANT SELECT, INSERT, UPDATE, DELETE ON plan\_table TO PUBLIC;** |



**TABLE 16-1** *Plan Table Columns*

**Generating an Execution Plan**

Once you have a plan table, you can use the EXPLAIN PLAN command to generate an execution plan for an SQL statement. The syntax for the EXPLAIN PLAN command is as follows:

EXPLAIN PLAN SET STATEMENT\_ID = *statement\_id* FOR *sql\_statement*;

where

Image *statement\_id* is the name you want to call the execution plan. This can be any alphanumeric text.

Image *sql\_statement* is the SQL statement you want to generate an execution plan for.

The following example generates the execution plan for a query that retrieves all rows from the customerstable (notice that the statement\_id is set to 'CUSTOMERS'):

**EXPLAIN PLAN SET STATEMENT\_ID = 'CUSTOMERS' FOR**

**SELECT customer\_id, first\_name, last\_name FROM customers;**

Explained

After the command completes, you may examine the execution plan stored in the plan table. You’ll see how to do that next.

Image

**NOTE**  
*The query in the* EXPLAIN PLAN *statement doesn’t return rows from the* customers *table. The*EXPLAIN PLAN *statement simply generates the execution plan that would be used if the query was run*.

**Querying the Plan Table**

For querying the plan table, I have provided an SQL\*Plus script named explain\_plan.sql in the SQLdirectory. The script prompts you for the statement\_id and then displays the execution plan for that statement.

The explain\_plan.sql script is as follows:

-- Displays the execution plan for the specified statement\_id

UNDEFINE v\_statement\_id;

SELECT

id ||

DECODE(id, 0, '', LPAD(' ', 2\*(level - 1))) || ' ' ||

operation || ' ' ||

options || ' ' ||

object\_name || ' ' ||

object\_type || ' ' ||

DECODE(cost, NULL, '', 'Cost = ' || position)

AS execution\_plan

FROM plan\_table

CONNECT BY PRIOR id = parent\_id

AND statement\_id = '&&v\_statement\_id'

START WITH id = 0

AND statement\_id = '&v\_statement\_id';

An execution plan is organized into a hierarchy of database operations similar to a tree; the details of these operations are stored in the plan table. The operation with an id of 0 is the root of the hierarchy, and all the other operations in the plan stem from this root. The query in the script retrieves the details of the operations, starting with the root operation and then navigating the tree from the root.

The following example shows how to run the explain\_plan.sql script to retrieve the 'CUSTOMERS' plan created earlier:

SQL> **@ c:\sql\_book\sql\explain\_plan.sql**

Enter value for v\_statement\_id: **CUSTOMERS**

old 12: statement\_id = '&&v\_statement\_id'

new 12: statement\_id = 'CUSTOMERS'

old 14: statement\_id = '&v\_statement\_id'

new 14: statement\_id = 'CUSTOMERS'

EXECUTION\_PLAN

----------------------------------------------

0 SELECT STATEMENT Cost = 3

1 TABLE ACCESS FULL CUSTOMERS TABLE Cost = 1

The operations shown in the EXECUTION\_PLAN column are executed in the following order:

Image The rightmost indented operation is executed first, followed by any parent operations above it.

Image For operations with the same indentation, the topmost operation is executed first, followed by any parent operations above it.

Each operation feeds its results back up the chain to its immediate parent operation, and the parent operation is then executed. In the EXECUTION\_PLAN column, the operation ID is shown on the far left. In the example execution plan, operation 1 is run first, with the results of that operation being passed to operation 0. The following example illustrates the ordering for a more complex example:

0 SELECT STATEMENT Cost = 6

1 MERGE JOIN Cost = 1

2 TABLE ACCESS BY INDEX ROWID PRODUCT\_TYPES TABLE Cost = 1

3 INDEX FULL SCAN PRODUCT\_TYPES\_PK INDEX (UNIQUE) Cost = 1

4 SORT JOIN Cost = 2

5 TABLE ACCESS FULL PRODUCTS TABLE Cost = 1

The order in which the operations are executed in this example is 3, 2, 5, 4, 1, and 0.

Now that you’ve seen the order in which operations are executed, it’s time to move onto what the operations actually do. The execution plan for the 'CUSTOMERS' query was

0 SELECT STATEMENT Cost = 3

1 TABLE ACCESS FULL CUSTOMERS TABLE Cost = 1

Operation 1 is run first, with the results of that operation being passed to operation 0. Operation 1 involves a full table scan—indicated by the string TABLE ACCESS FULL—on the customers table. Here’s the original command used to generate the 'CUSTOMERS' query:

EXPLAIN PLAN SET STATEMENT\_ID = 'CUSTOMERS' FOR

SELECT customer\_id, first\_name, last\_name FROM customers;

A full table scan is performed because the SELECT statement specifies that all the rows from the customerstable are to be retrieved.

The total cost of the query is three work units, as indicated in the cost part shown to the right of operation 0 in the execution plan (0 SELECT STATEMENT Cost = 3). A work unit is the amount of processing the software has to do to perform a given operation. The higher the cost, the more work the database software has to do to complete the SQL statement.

Image

**NOTE**  
*If you’re using a version of the database prior to Oracle Database 10*g, *then the output for the overall statement cost may be blank. That’s because earlier database versions don’t automatically collect table statistics. In order to gather statistics, you have to use the* ANALYZE *command. You’ll learn how to do that later in the section "Gathering Table Statistics.“*

**Execution Plans Involving Table Joins**

Execution plans for queries with table joins are more complex. The following example generates the execution plan for a query that joins the products and product\_types tables:

**EXPLAIN PLAN SET STATEMENT\_ID = 'PRODUCTS' FOR**

**SELECT p.name, pt.name**

**FROM products p, product\_types pt**

**WHERE p.product\_type\_id = pt.product\_type\_id;**

The execution plan for this query is shown in the following example:

**@ c:\sql\_book\sql\explain\_plan.sql**

Enter value for v\_statement\_id: **PRODUCTS**

EXECUTION\_PLAN

----------------------------------------------------------------

0 SELECT STATEMENT Cost = 6

1 MERGE JOIN Cost = 1

2 TABLE ACCESS BY INDEX ROWID PRODUCT\_TYPES TABLE Cost = 1

3 INDEX FULL SCAN PRODUCT\_TYPES\_PK INDEX (UNIQUE) Cost = 1

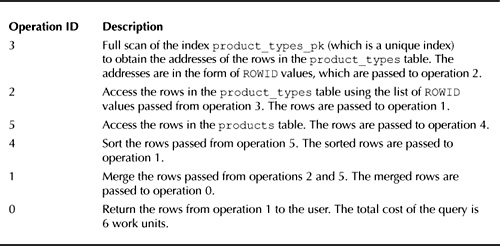
4 SORT JOIN Cost = 2

5 TABLE ACCESS FULL PRODUCTS TABLE Cost = 1

Image

**NOTE**  
*If you run the example, you may get a slightly different execution plan depending on the version of the database you are using and on the settings of the parameters in the database’s* init.ora*configuration file*.

The previous execution plan is more complex, and you can see the hierarchical relationships between the various operations. The execution order of the operations is 3, 2, 5, 4, 1, and 0. [Table 16-2](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch16.html#table_16-2) describes each operation in the order they are performed.



**TABLE 16-2** *Execution Plan Operations*

**Gathering Table Statistics**

If you’re using a version of the database prior to Oracle Database 10*g* (such as 9*i*), then you’ll have to gather table statistics yourself using the ANALYZE command. By default, if no statistics are available then rule-based optimization is used. Rule-based optimization isn’t usually as good as cost-based optimization.

The following examples use the ANALYZE command to gather statistics for the products and product\_types tables:

**ANALYZE TABLE products COMPUTE STATISTICS;**

**ANALYZE TABLE product\_types COMPUTE STATISTICS;**

Once the statistics have been gathered, cost-based optimization will be used rather than rule-based optimization.

**Comparing Execution Plans**

By comparing the total cost shown in the execution plan for different SQL statements, you can determine the value of tuning your SQL. In this section, you’ll see how to compare two execution plans and see the benefit of using EXISTS rather than DISTINCT (a tip I gave earlier). The following example generates an execution plan for a query that uses EXISTS:

**EXPLAIN PLAN SET STATEMENT\_ID = 'EXISTS\_QUERY' FOR**

**SELECT product\_id, name**

**FROM products outer**

**WHERE EXISTS**

**(SELECT 1**

**FROM purchases inner**

**WHERE inner.product\_id = outer.product\_id);**

The execution plan for this query is shown in the following example:

**@ c:\sql\_book\sql\explain\_plan.sql**

Enter value for v\_statement\_id: **EXISTS\_QUERY**

EXECUTION\_PLAN

------------------------------------------------------------

0 SELECT STATEMENT Cost = 4

1 MERGE JOIN SEMI Cost = 1

2 TABLE ACCESS BY INDEX ROWID PRODUCTS TABLE Cost = 1

3 INDEX FULL SCAN PRODUCTS\_PK INDEX (UNIQUE) Cost = 1

4 SORT UNIQUE Cost = 2

5 INDEX FULL SCAN PURCHASES\_PK INDEX (UNIQUE) Cost = 1

As you can see, the total cost of the query is 4 work units. The next example generates an execution plan for a query that uses DISTINCT:

**EXPLAIN PLAN SET STATEMENT\_ID = ’DISTINCT\_QUERY' FOR**

**SELECT DISTINCT pr.product\_id, pr.name**

**FROM products pr, purchases pu**

**WHERE pr.product\_id = pu.product\_id;**

The execution plan for this query is shown in the following example:

**@ c:\sql\_book\sql\explain\_plan.sql**

Enter value for v\_statement\_id: **DISTINCT\_QUERY**

EXECUTION\_PLAN

--------------------------------------------------------------

0 SELECT STATEMENT Cost = 5

1 HASH UNIQUE Cost = 1

2 MERGE JOIN Cost = 1

3 TABLE ACCESS BY INDEX ROWID PRODUCTS TABLE Cost = 1

4 INDEX FULL SCAN PRODUCTS\_PK INDEX (UNIQUE) Cost = 1

5 SORT JOIN Cost = 2

6 INDEX FULL SCAN PURCHASES\_PK INDEX (UNIQUE) Cost = 1

The cost for the query is 5 work units. This query is more costly than the earlier query that used EXISTS (that query had a cost of only 4 work units). These results prove it is better to use EXISTS than DISTINCT.

**PASSING HINTS TO THE OPTIMIZER**

You can pass hints to the optimizer. A hint is an optimizer directive that influences the optimizer’s choice of execution plan. The correct hint may improve the performance of an SQL statement. You can check the effectiveness of a hint by comparing the cost in the execution plan of an SQL statement with and without the hint.

In this section, you’ll see an example query that uses one of the more useful hints: the FIRST\_ROWS(*n*) hint. The FIRST\_ROWS(*n*) hint tells the optimizer to generate an execution plan that will minimize the time taken to return the first *n* rows in a query. This hint can be useful when you don’t want to wait around too long before getting *some* rows back from your query, but you still want to see all the rows.

The following example generates an execution plan for a query that uses FIRST\_ROWS (2); notice that the hint is placed within the strings /\*+ and \*/:

**EXPLAIN PLAN SET STATEMENT\_ID = 'HINT' FOR**

**SELECT /\*+ FIRST\_ROWS(2) \*/ p.name, pt.name**

**FROM products p, product\_types pt**

**WHERE p.product\_type\_id = pt. product\_type\_id;**

Image

**CAUTION**  
*Your hint must use the* exact *syntax shown—otherwise, the hint will be ignored. The syntax is:* /\*+*followed by one space, the hint, followed by one space, and* \*/.

The execution plan for this query is shown in the following example; notice that the cost is 4 work units:

**@ c:\sql\_book\sql\explain\_plan.sql**

Enter value for v\_statement\_id: **HINT**

EXECUTION\_PLAN

------------------------------------------------------------------

0 SELECT STATEMENT Cost = 4

1 NESTED LOOPS

2 NESTED LOOPS Cost = 1

3 TABLE ACCESS FULL PRODUCTS TABLE Cost = 1

4 INDEX UNIQUE SCAN PRODUCT\_TYPES\_PK INDEX (UNIQUE) Cost = 2

5 TABLE ACCESS BY INDEX ROWID PRODUCT\_TYPES TABLE Cost = 2

The next example generates an execution plan for the same query without the hint:

**EXPLAIN PLAN SET STATEMENT\_ID = 'NO\_HINT' FOR**

**SELECT p.name, pt.name**

**FROM products p, product\_types pt**

**WHERE p.product\_type\_id = pt. product\_type\_id;**

The execution plan for the query is shown in the following example; notice the cost is 6 work units (higher than the query with the hint):

**@ c:\sql\_book\sql\explain\_plan.sql**

Enter value for v\_statement\_id: **NO\_HINT**

EXECUTION\_PLAN

----------------------------------------------------------------

0 SELECT STATEMENT Cost = 6

1 MERGE JOIN Cost = 1

2 TABLE ACCESS BY INDEX ROWID PRODUCT\_TYPES TABLE Cost = 1

3 INDEX FULL SCAN PRODUCT\_TYPES\_PK INDEX (UNIQUE) Cost = 1

4 SORT JOIN Cost = 2

5 TABLE ACCESS FULL PRODUCTS TABLE Cost = 1

These results show that the inclusion of the hint reduces the cost of running the query by 2 work units.

There are many hints that you can use, and this section has merely given you a taste of the subject.

**ADDITIONAL TUNING TOOLS**

In this final section, I’ll mention some other tuning tools. Full coverage of these tools is beyond the scope of this book. You can read the *Oracle Database Performance Tuning Guide*, published by Oracle Corporation, for full details of the tools mentioned in this section and for a comprehensive list of hints.

**Oracle Enterprise Manager Diagnostics Pack**

The Oracle Enterprise Manager Diagnostics Pack captures operating system, middle tier, and application performance data, as well as database performance data. The Diagnostics Pack analyzes this performance data and displays the results graphically. A database administrator can also configure the Diagnostics Pack to alert them immediately of performance problems via e-mail or page. Oracle Enterprise Manager also includes software guides to help resolve performance problems.

**Automatic Database Diagnostic Monitor**

The Automatic Database Diagnostic Monitor (ADDM) is a self-diagnostic module built into the Oracle database software. ADDM enables a database administrator to monitor the database for performance problems by analyzing system performance over a long period of time. The database administrator can view the performance information generated by ADDM in Oracle Enterprise Manager. When ADDM finds performance problems, it will suggest solutions for corrective action. Some example ADDM suggestions include

Image Hardware changes—for example, adding CPUs to the database server

Image Database configuration—for example, changing the database initialization parameter settings

Image Application changes—for example, using the cache option for sequences or using bind variables

Image Use other advisors—for example, running the SQL Tuning Advisor and SQL Access Advisor on SQL statements that are consuming the most database resources to execute

You’ll learn about the SQL Tuning Advisor and SQL Access Advisor next.

**SQL Tuning Advisor**

The SQL Tuning Advisor allows a developer or database administrator to tune an SQL statement using the following items:

Image The text of the SQL statement

Image The SQL identifier of the statement (obtained from the V$SQL\_PLAN view, which is one of the views available to a database administrator)

Image The range of snapshot identifiers

Image The SQL Tuning Set name

An SQL Tuning Set is a set of SQL statements with their associated execution plan and execution statistics. SQL Tuning Sets are analyzed to generate SQL Profiles that help the optimizer to choose the optimal execution plan. SQL Profiles contain collections of information that enable optimization of the execution plan.

**SQL Access Advisor**

The SQL Access Advisor provides a developer or database administrator with performance advice on materialized views, indexes, and materialized view logs. The SQL Access Advisor examines space usage and query performance and recommends the most cost-effective configuration of new and existing materialized views and indexes.

**SUMMARY**

In this chapter, you have learned the following:

Image Tuning is the process of making your SQL statements run faster.

Image The optimizer is a subsystem of the Oracle database software that generates an execution plan, which is a set of operations used to perform a particular SQL statement.

Image Hints may be passed to the optimizer to influence the generated execution plan for an SQL statement.

Image There are a number of additional software tools a database administrator can use to tune the database.