CHAPTER 19

Tuning SQL Statements

In This Chapter

Using the Explain Plan feature

Using SQL*Plus autotrace

Using SQL Trace and TKPROF

If you want to get maximum performance from your applications, you need to tune your SQL statements. Tuning a SQL statement means discovering the execution plan that Oracle is using. Once you know the execution plan, you can attempt to improve it.

You can discover the execution plan for a statement in at least three ways. One is to issue an `EXPLAIN PLAN` statement from SQL*Plus. Another alternative is to use the SQL*Plus autotrace feature, and still another alternative is to use Oracle's SQL Trace feature. When you use SQL Trace, you can log statistics about statements as they are being executed, which is invaluable if you are trying to fix a poorly performing process.

You can attempt to improve the performance of a query in many ways. You can create indexes, you can increase the size of the buffer cache, and you can use optimizer hints. This chapter concentrates on the use of optimizer hints. *Hints* are instructions to the Oracle optimizer that are buried within your statement. You can use hints to control virtually any aspect of statement execution.

Using the Explain Plan Feature

Oracle's Explain Plan feature allows you to discover the execution plan for a SQL statement. You do this by using a SQL statement, not surprisingly named `EXPLAIN PLAN`, to which you append the query of interest. Consider this example:

```
EXPLAIN PLAN
SET STATEMENT_ID = 'q1'
FOR
SELECT *
FROM aquatic_animal;
```

After issuing this statement, you can query a special table known as the *plan table* to find the execution plan that Oracle intends to use for the query. Here's an example of an extremely simple execution plan, which happens to be the plan that Oracle would use for the preceding query:

```
TABLE ACCESS FULL AQUATIC_ANIMAL
```

There's not much to this plan, but it tells you that Oracle intends to read all the rows in the `AQUATIC_ANIMAL` table to satisfy your query.

The `EXPLAIN PLAN` statement is most often used from SQL*Plus, or from SQLPlus Worksheet. Before you can use `EXPLAIN PLAN`, you need to create a plan table to hold the results. You also need to know how to query that table.

Creating the plan table

When you issue an `EXPLAIN PLAN` statement, Oracle doesn't display the results on the screen. In fact, the Oracle database software has no way to directly write data to the display at all. Instead, when you issue an `EXPLAIN PLAN` statement, Oracle writes the results to a table known as the *plan table*, and it's up to you to create it.

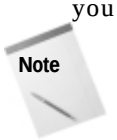
The easiest way to create a plan table is to use the Oracle-supplied script named utlxplan.sql. You'll find that script in your $ORACLE_HOME/rdbms/admin directory. Listing 19-1 shows utlxplan.sql being invoked from SQL*Plus.

Listing 19-1: **Creating the plan table**

```
SQL> @e:\oracle\ora81\rdbms\admin\utlxplan

Table created.

SQL> DESCRIBE plan_table

 Name                 Null?    Type
 -------------------- -------- -----------------
 STATEMENT_ID                  VARCHAR2(30)
 TIMESTAMP                     DATE
 REMARKS                       VARCHAR2(80)
 OPERATION                     VARCHAR2(30)
 OPTIONS                       VARCHAR2(30)
 OBJECT_NODE                   VARCHAR2(128)
 OBJECT_OWNER                  VARCHAR2(30)
 OBJECT_NAME                   VARCHAR2(30)
 OBJECT_INSTANCE               NUMBER(38)
 OBJECT_TYPE                   VARCHAR2(30)
 OPTIMIZER                     VARCHAR2(255)
 SEARCH_COLUMNS                NUMBER
```

```
ID                             NUMBER(38)
PARENT_ID                      NUMBER(38)
POSITION                       NUMBER(38)
COST                           NUMBER(38)
CARDINALITY                    NUMBER(38)
BYTES                          NUMBER(38)
OTHER_TAG                      VARCHAR2(255)
PARTITION_START                VARCHAR2(255)
PARTITION_STOP                 VARCHAR2(255)
PARTITION_ID                   NUMBER(38)
OTHER                          LONG
DISTRIBUTION                   VARCHAR2(30)
```

The name of the table created by the script is `PLAN_TABLE`. This is also the default table name used by the `EXPLAIN PLAN` statement. You can name the plan table something else if you want to, but then you have to supply that name every time you explain a plan.



The structure of the plan table sometimes changes from one release of Oracle to the next. Consequently, when you upgrade, you may need to drop and re-create your plan tables.

It's normal to have more than one plan table in a database. In fact, it's quite typical for every user using the `EXPLAIN PLAN` feature to have his or her own plan table in his or her own schema.

Explaining a query

Having created a plan table, you are now in a position to explain the execution plan for a query. Say that you have the following `SELECT` statement:

```
SELECT a.id_no, a.animal_name, a.tank_no,
       NVL(c.caretaker_name, t.chief_caretaker_name)
FROM aquatic_animal a, tank t, caretaker c
WHERE a.tank_no = t.tank_no
AND c.caretaker_name (+) = t.chief_caretaker_name
ORDER BY a.tank_no, a.animal_name;
```

This statement contains two joins, one an inner join, and one an outer join. It also contains an `ORDER BY` clause, which results in a sort. If you want to know how Oracle is going to go about executing this query, you can use an `EXPLAIN PLAN` statement that looks like this:

```
EXPLAIN PLAN
SET STATEMENT_ID = 'user_supplied_name'
FOR query;
```

You're basically appending the keywords `EXPLAIN PLAN ... FOR` onto the front of your query and executing the result. You need to give the results a name so that you can query them later. That's what the `SET STATEMENT_ID` clause is for. Whatever text you supply as a statement ID is stored with the results in the plan table.

Explaining a plan doesn't cause the SQL statement to be executed. Oracle determines the execution plan, nothing more.

Listing 19-2 shows `EXPLAIN PLAN` being used.

Listing 19-2: **Using the EXPLAIN PLAN statement**

```
SQL> DELETE
  2  FROM plan_table
  3  WHERE statement_id = 'caretakers';

6 rows deleted.

SQL>
SQL> EXPLAIN PLAN
  2  SET STATEMENT_ID = 'caretakers'
  3  FOR
  4  SELECT a.id_no, a.animal_name, a.tank_no,
  5         NVL(c.caretaker_name, t.chief_caretaker_name)
  6  FROM aquatic_animal a, tank t, caretaker c
  7  WHERE a.tank_no = t.tank_no
  8  AND c.caretaker_name (+) = t.chief_caretaker_name
  9  ORDER BY a.tank_no, a.animal_name;

Explained.
```

The delete is necessary because `EXPLAIN PLAN` always adds to the plan table. If any records happen to exist with a statement ID of '`caretakers`', they play havoc with the `EXPLAIN PLAN` results. It's safest to do a delete first, to avoid any possibility of duplication.

If you reuse a statement ID without deleting the plan table entries from the previous time, you'll get bizarre results when you query the plan table. This is especially true if you query the plan table using a `CONNECT BY` query. Instead of getting a five-line plan, for example, you might get 50+ lines, many of them duplicates of each other.

After you've explained a plan, you can query the PLAN TABLE to see the details of that plan.

Showing the execution plan for a query

The records in the plan table are related to each other hierarchically. Each record has an ID column and a PARENT_ID column. You execute a given SQL statement by performing a series of operations. Each operation, in turn, may consist of one or more operations. The most deeply nested operations are executed first. They, in turn, feed results to their parents. This process continues until only one result is left — the result of the query — which is returned to you.

Using the Plan Table Query

The three most significant columns in the plan table are named OPERATION, OPTIONS, and OBJECT_NAME. For each step, these tell you which operation is going to be performed and which object is the target of that operation.

You can use the following SQL query to display an execution plan once it has been generated:

```
SELECT id,
       LPAD(' ', 2*(level-1)) || operation
       || ' ' || options
       || ' ' || object_name
       || ' ' ||
       DECODE(id, 0, 'Cost = ' || position)
          step_description
FROM plan_table
START WITH id = 0 AND statement_id = 'statement_id'
CONNECT BY prior id = parent_id
AND statement_id = 'statement_id'
ORDER BY id, position;
```

You use the CONNECT BY clause to link each operation with its parent. The LPAD business is for indention so that if an operation has child rows, those rows will be indented underneath the parent.

Showing the Plan for the Caretakers Statement

Listing 19-3 shows the plan table being queried for the results of the EXPLAIN PLAN on the caretakers statement.

Listing 19-3: **Querying the plan table**

```
SQL> SELECT id,
  2         LPAD(' ', 2*(level-1)) || operation
  3         || ' ' || options
  4         || ' ' || object_name
  5         || ' ' ||
  6         DECODE(id, 0, 'Cost = ' || position)
  7            step_description
  8  FROM plan_table
  9  START WITH id = 0 AND statement_id = 'caretakers'
 10  CONNECT BY prior id = parent_id
 11  AND statement_id = 'caretakers'
 12  ORDER BY id, position;

  ID STEP_DESCRIPTION
---- --------------------------------------------------
   0 SELECT STATEMENT   Cost = 9
   1   SORT ORDER BY
   2     HASH JOIN
   3       NESTED LOOPS OUTER
   4         TABLE ACCESS FULL TANK
   5         INDEX UNIQUE SCAN CARETAKER_PK
   6       TABLE ACCESS FULL AQUATIC_ANIMAL

7 rows selected.
```

To interpret this plan, you need to read your way outwards from the most deeply nested operations. For example:

1. The most deeply nested operations are 4 and 5. The parent operation is a nested loop (3). This means that Oracle will read all rows in the `TANK` table (4), and for each tank row, it will read the corresponding `CARETAKER` row (5). Oracle accesses the `CARETAKER` table via the primary key index named `CARETAKER_PK`.
2. The results of the join between `TANK` and `CARETAKER` are fed into a hash join process (2) that pulls in the corresponding rows from the `AQUATIC_ANIMAL` table. A full tablescan will be done on the `AQUATIC_ANIMAL` table, meaning that all the rows will be accessed.
3. The results of the hash join (2) are fed into a sort operation (1). This orders the results as requested by the query's `ORDER BY` clause. In this case, the results will be ordered by tank number and animal name.
4. The results of the sort (1) are fed back as the results of the `SELECT` statement (0), and are what you ultimately see.

The cost that you see, 9 in the previous example, is an arbitrary number used to give you some idea of the relative amount of I/O and CPU space required to execute a query. The *cost* of a query is derived from the `POSITION` column in the row containing operation `ID` zero. The plan table query shown here uses the `DECODE` function to display the cost when that row is encountered.

By itself, the number means nothing. You have to compare it to the cost for alternate execution plans. A plan with a cost of 9 will consume more or less half the resources as a plan with a cost of 18.

The execution plan cost comes into play only when the cost-based optimizer is being used. Use the rule-based optimizer, and your cost will always be null.

Understanding Execution Plan Operations

To understand an execution plan like the one shown in the previous section, you need to understand the meanings of the different operations together with their options. Table 19-1 provides brief descriptions of each combination.

Table 19-1
Execution Plan Operations

Operation + Option	*Description*
`AND-EQUAL`	Executes two child operations and returns only those rows that are returned by both
`BITMAP CONVERSION TO ROWIDS`	Converts a bitmap into a set of `ROWID`s
`BITMAP CONVERSION FROM ROWIDS`	Converts a set of `ROWID`s into a bitmap
`BITMAP CONVERSION COUNT`	Counts the number of entries in a bitmap
`BITMAP INDEX SINGLE VALUE`	Retrieves the bitmap that corresponds to a specific value in the column
`BITMAP INDEX RANGE SCAN`	Returns the bitmaps that correspond to a range of values in a column
`BITMAP INDEX FULL SCAN`	Scans a bitmapped index
`BITMAP MERGE`	Merges two bitmaps into one
`BITMAP MINUS`	Subtracts one bitmap from another
`BITMAP OR`	Merges two bitmaps into one

Continued

Table 19-1 *(continued)*

Operation + Option	*Description*
CONNECT BY	Retrieves rows hierarchically
CONCATENATION	Combines multiple rowsets into one
COUNT	Counts rows
FILTER	Filters a set of rows based on some condition in the WHERE clause
FIRST ROW	Returns only the first row of a query's results
FOR UPDATE	Indicates that the retrieved rows will be locked for subsequent updating
HASH JOIN	Joins two tables
INDEX UNIQUE	Uses an index to look up a specific value
INDEX RANGE SCAN	Uses an index to look up a range of values
INLIST ITERATOR	Performs an operation once for each value listed in an IN list
INTERSECTION	Looks at two rowsets and returns only those rows found in both
MERGE JOIN	Joins two tables
MERGE JOIN OUTER	Performs an outer join of two tables
MINUS	Subtracts one rowset from another
NESTED LOOPS	Executes one child operation for each row returned by another
PARTITION SINGLE	Executes an operation on one partition of a table or an index
PARTITION ITERATOR	Performs an operation on a list of partitions
PARTITION ALL	Performs an operation on all partitions of a table or an index
PARTITION INLIST	Performs an operation on partitions associated with an IN clause
PROJECTION	Returns a single set of records for a set of queries
REMOTE	Performs an operation on a remote database
SEQUENCE	Retrieves a value from an Oracle sequence
SORT AGGREGATE	Applies an aggregate function to a rowset
SORT UNIQUE	Sorts rows, eliminating duplicates
SORT GROUP BY	Sorts rows into groups
SORT JOIN	Sorts rows in preperation for a join
SORT ORDER BY	Sorts rows as specified by a query's ORDER BY clause
TABLE ACCESS FULL	Reads every row in a table

Operation + Option	*Description*
TABLE ACCESS CLUSTER	Reads every row in a table that matches a given cluster key
TABLE ACCESS HASH	Reads every row in a table that matches a given hash cluster key
TABLE ACCESS BY ROWID	Retrieves a row from a table using the ROWID
UNION	Performs a SQL union
VIEW	Executes the query for a view

Hinting for a better plan

Now that you know how Oracle is planning to execute the query, you can think about ways in which to improve performance. One way that you can change an execution plan that you don't like is to add optimizer *hints* to the query. These hints take the form of specially formatted comments and tell the optimizer how *you* want the query to be executed. They aren't really hints, either, even though Oracle refers to them as such. They're more like commands. It's true that Oracle will ignore hints that are contradictory or that are impossible to carry out, but you'll never see Oracle ignore a hint when it can be carried out.

Looking at a Hint Example

Recall that the execution plan for the caretakers query shown earlier looked like this:

```
0 SELECT STATEMENT   Cost = 9
1   SORT ORDER BY
2     HASH JOIN
3       NESTED LOOPS OUTER
4         TABLE ACCESS FULL TANK
5         INDEX UNIQUE SCAN CARETAKER_PK
6       TABLE ACCESS FULL AQUATIC_ANIMAL
```

Notice that an index retrieval is done on the CARETAKER table when it is joined to the TANK table. Suppose that, based on your knowledge of the data, a full tablescan would me more efficient than using an index. You could force that behavior by adding a hint to the query. Here's an example where three hints are used to force full tablescans on all the tables:

```
SELECT /*+ full(a) full(t) full(c) */
       a.id_no, a.animal_name, a.tank_no,
       NVL(c.caretaker_name, t.chief_caretaker_name)
FROM aquatic_animal a, tank t, caretaker c
WHERE a.tank_no = t.tank_no
AND c.caretaker_name (+) = t.chief_caretaker_name
ORDER BY a.tank_no, a.animal_name;
```

The three hints are: `full(a)`, `full(t)`, and `full(c)`. These tell Oracle to perform full tablescans on each of the three tables. The resulting execution plan looks like this:

```
0 SELECT STATEMENT   Cost = 11
1   SORT ORDER BY
2     HASH JOIN
3       HASH JOIN OUTER
4         TABLE ACCESS FULL TANK
5         TABLE ACCESS FULL CARETAKER
6       TABLE ACCESS FULL AQUATIC_ANIMAL
```

Voila! No more indexes. Notice that your cost has gone up from 9 to 11. This is Oracle's interpretation of the query cost. It may not reflect reality. The only way to be sure is through some real-world testing of the query.

Examining Hint Syntax

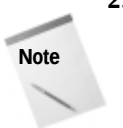
When you place a hint in a SQL statement, the comment needs to take a special form, and it needs to immediately follow the verb. For example:

```
SELECT /*+ hint comment hint hint comment ...*/
...
```

Notice the plus (+) sign immediately following the start of the comment. That plus sign tells Oracle that the comment contains hints. The comment may contain several hints, and you may have comments interspersed with your hints. You should separate hints from each other, and from any nonhint text, by at least one space.

Keep the following rules in mind when writing hints:

1. The comment with the hints must immediately follow the `SELECT`, `INSERT`, `UPDATE`, or `DELETE` keyword.
2. For Oracle to recognize the hints, the comment must start with `/*+`.



As an alternative, you can use `--+` to start a comment containing hints. If you do this, the comment continues until the end of the line is reached. In practice, almost everyone uses `/*+ ... */`.

3. Several hints allow you to reference table names and/or index names. Table and index names are always enclosed within parentheses.
4. If you alias a table in your query, you must use that alias in any hints that refer to the table.
5. If you supply contradictory hints, Oracle will ignore at least one of them.
6. If your hints refer to indexes that don't exist, Oracle will ignore them.
7. If you want to control the execution of a subquery, you must place a hint in that subquery. Hints in the main query refer to the main query. Hints in a subquery refer to the subquery.

8. If you write a hint incorrectly, Oracle will ignore it.
9. Oracle never returns error messages for badly written hints. Hints are embedded within comments, and to comply with ANSI standards, error messages can't be returned for comments.

The last item, number 9, is particularly important. If you get a hint wrong, don't start the comment correctly, or put the comment in the wrong place, you won't get an error message. Therefore, you should always perform an explain plan on any queries containing hints to be sure that Oracle is recognizing those hints.

Understanding the Available Hints

Table 19-2 describes each of the hints that Oracle supports.

Table 19-2
Oracle's Optimizer Hints

Hint	*Description*
`ALL_ROWS`	Produces an execution plan that optimizes overall resource usage.
`AND_EQUAL(`*`table_name index_name index name...`*`)`	Specifies to access a table by scanning two or more indexes and merging the results. You must specify at least two index names with this hint.
`APPEND`	Specifies not to reuse any free space that may be available in any extents allocated to a table. This hint applies only to `INSERT` statements.
`CACHE (`*`table_name`*`)`	Specifies to keep data from the named table in memory as long as possible.
`CHOOSE`	Specifies to use the cost-based optimizer if statistics exist for any of the tables involved.
`CLUSTER(`*`table_name`*`)`	Specifies to access the named table by scanning the cluster. This hint is valid only for clustered tables.
`DRIVING_SITE (`*`table_name`*`)`	Specifies which database to use as the driving site when joining tables from two different databases. The driving site will be the database in which the named table resides.
`FIRST_ROWS`	Produces an execution plan that optimizes for a quick initial response.
`FULL(`*`table_name`*`)`	Specifies to access the named table by reading all the rows.

Continued

Table 19-2 *(continued)*

Hint	*Description*
HASH(*table_name*)	Specifies to do a hash scan of the named table. This hint is valid only for hash-clustered tables.
HASH_AJ(*table_name*)	Specifies to do a hash anti-join of the specified table.
INDEX(*table_name* [*index_name...*])	Specifies to access the named table through an index. You may optionally specify a list of indexes from which to choose.
INDEX_ASC(*table_name* [*index_name...*])	Specifies an index to scan, the same as INDEX, but also specifies to scan the index in ascending order.
INDEX_COMBINE (*table_name* [*index_name...*])	Specifies to access the table using a combination of two indexes. You may optionally supply a list of indexes from which to choose.
INDEX_DESC(*table_name* [*index_name...*])	Specifies an index to scan, the same as INDEX, but also specifies to scan the index in descending order.
INDEX_FFS(*table_name* [*index_name...*])	Specifies to access rows in the table via a fast full index scan. You may optionally supply a list of indexes that Oracle can choose from.
MERGE_AJ(*table_name*)	Specifies to resolve a NOT IN subquery for the named table by performing a merge anti-join.
NO_MERGE	Specifies not to merge a query into a view's query. This hint applies only to queries against views.
NO_PARALLEL(*table_name*)	Specifies not to use any parallel processing features when accessing the named table.
NOAPPEND	Specifies to use free space in extents currently allocated to a table. This hint applies only to INSERT statements.
NOCACHE(*table_name*)	Specifies to remove data from memory from the named table as quickly as possible.
ORDERED	Specifies to join tables from left to right, in the same order in which they are listed in the FROM clause of the query.
PARALLEL(*table_name* [, *degree*[, *num_instances*]])	Specifies the degree of parallelism, and optionally the number of instances, to use when accessing the named table.
PARALLEL_INDEX (*table_name*, *index_name* [,*degree* [,*num_instances*]])	Specifies to access the named table via an index, and to use parallel processing features to scan that index. You may optionally specify the degree of parallelism and the number of instances to use.

Hint	*Description*
PUSH_SUBQ	Specifies to evaluate subqueries as soon as possible during query execution.
ROWID(*table_name*)	Specifies to access the named table using ROWIDs.
RULE	Specifies to use the rule-based optimizer.
STAR	Specifies to use a star query execution plan, if at all possible.
STAR_TRANSFORMATION	Specifies to transform the query into a star query, if at all possible.
USE_CONCAT	Specifies to convert a query with conditions into two or more queries unioned together.
USE_HASH(*table_name*)	Specifies to use a hash join whenever the named table is joined to any other table.
USE_MERGE(*table_name*)	Specifies to use a merge join whenever the named table is joined to any other table.
USE_NL(*table_name*)	Specifies to use a nested loop when joining the named table to any other table. The other table will always be the driving table.

Using SQL*Plus Autotrace

If you're using SQL*Plus release 3.3 or higher, you can take advantage of the autotrace feature to have queries explained automatically. The process is simple. You turn autotrace on using the SET command, and you issue the query as you normally would. SQL*Plus will execute the query and display the execution plan following the results. Listing 19-4 shows an example.

Listing 19-4: **Using the autotrace feature**

```
SQL> SET AUTOTRACE ON EXPLAIN
SQL> SELECT animal_name
  2  FROM aquatic_animal
  3  ORDER BY animal_name;

ANIMAL_NAME
------------------------------
Batty
Bopper
Flipper
Nosey
```

Continued

Listing 19-4: *(continued)*

```
Paintuin
Rascal
Shorty
Skipper
Snoops
Squacky

10 rows selected.


Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=3 Card=10
             Bytes=170)
   1    0   SORT (ORDER BY) (Cost=3 Card=10 Bytes=170)
   2    1     TABLE ACCESS (FULL) OF 'AQUATIC_ANIMAL' (Cost=1
              Card=10 Bytes=170)
```

If you leave off the keyword `EXPLAIN`, and issue just the command `SET AUTOTRACE ON`, SQL*Plus will display execution statistics in addition to the execution plan. To do this, you must either be the DBA or have the PLUSTRACE role.

Using autotrace is convenient because you don't need to manually query the plan table. SQL*Plus does it for you automatically. However, SQL*Plus does execute the query. If a query generates a lot of I/O and consumes a lot of CPU, you won't want to kick it off just to see the execution plan. The `EXPLAIN PLAN` statement would be a better choice. If you don't mind executing the query but don't want to see the results, use the `TRACEONLY` setting. Consider this example:

```
SQL> SET AUTOTRACE TRACEONLY EXPLAIN
SQL> SELECT animal_name
  2  FROM aquatic_animal
  3  ORDER BY animal_name;

Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=3 Card=10 Bytes=170)
   1    0   SORT (ORDER BY) (Cost=3 Card=10 Bytes=170)
   2    1     TABLE ACCESS (FULL) OF 'AQUATIC_ANIMAL' (Cost=1 Card=10
           Bytes=170)
```

Even when you use `TRACEONLY`, SQL*Plus still sends the query to the database where it is executed. The `TRACEONLY` setting just prevents SQL*Plus from pulling back the results.

When you are through using autotrace, you can turn the feature off by issuing the `SET AUTOTRACE OFF` command.

Using SQL Trace and TKPROF

Oracle includes a facility known as SQL Trace that is extremely useful for diagnosing performance problems on running systems. It logs information to an operating system file for all the queries executed by a specific session, or by all sessions. Later, you can review that information, find out which queries are consuming the most CPU or generating the most I/O, and take some corrective action. SQL Trace returns the following information for each SQL statement:

- A count of the times that the statement was executed
- The total CPU and elapsed time used by the statement
- The total CPU and elapsed times for the parse, execute, and fetch phases of the statement's execution
- The total number of physical reads triggered by the statement
- The total number of logical reads triggered by the statement
- The total number of rows processed by the statement

The information on physical I/O and CPU time is most helpful when it comes to identifying statements that are causing performance problems.

Taking care of prerequisites

Before you can use SQL Trace, you need to take care of some prerequisites. Three initialization parameters affect how SQL Trace operates. One of them points to the location in which Oracle writes the trace files. This is important because you need to know where to find a trace file after you generate it. You also need to know how Oracle names these files.

Checking Initialization Parameters

The three initialization parameters that affect SQL tracing are the following:

- `TIMED_STATISTICS` — Controls whether Oracle tracks CPU and elapsed time for each statement. Always set this parameter to `TRUE`. The overhead for that is minimal, and the value of the timing information is well worth it.
- `MAX_DUMP_FILE_SIZE` — Controls the maximum size of the trace file generated by the SQL Trace facility. These files get very large very fast, so Oracle allows you to limit their size.

- ✦ USER_DUMP_DEST — Points to the directory in which trace files are created. This parameter is important when you want to find trace files.

You can control the TIMED_STATISTICS parameter at the session level rather than at the database level. To turn timed statistics on for your session, issue this command:

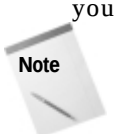
```
ALTER SESSION SET TIMED_STATISTICS = TRUE;
```

You can't set the maximum file size at the session level. You must set it in the parameter file for the instance. You can specify the maximum file size in either bytes or operating- system blocks. If you specify the size using a raw number, Oracle interprets it to mean blocks. If you supply a suffix such as K or M, Oracle interprets it as kilobytes or megabytes. For example:

MAX_DUMP_FILE_SIZE=100	100 blocks
MAX_DUMP_FILE_SIZE=100K	100KB
MAX_DUMP_FILE_SIZE=100M	100MB

If you are tracing a long process and the size of the trace file reaches the limit specified by MAX_DUMP_FILE_SIZE, then Oracle will silently stop tracing. The process will continue to run.

The USER_DUMP_DEST parameter points you to the directory where the trace files are created. Knowing its value is critical to the task of finding the trace files that you generate.

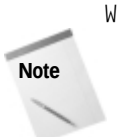


If you change any of these parameters in the database parameter file, you will need to stop and restart the instance for those changes to take effect.

Finding Your Trace Files

To find your trace files, you need to know the following: which directory they were written to, and their names. The USER_DUMP_DEST initialization parameter points to the directory. You can check the value of this parameter by looking in the database parameter file or by issuing this query:

```
SELECT value
FROM v$parameter
WHERE name = 'user_dump_dest';
```



If you're not the DBA, you may not have access to the v$parameter view.

Finding the directory is the easy part. Figuring out which trace file is yours is a bit more difficult. Oracle generates trace file names automatically, and the names are based on numbers that aren't always easy to trace back to a session. A typical trace file name would be ORA00230.TRC.

So how do you find your trace file? One way is by looking at the timestamp. Write down the time of day when you enable tracing, and also the time at which you finish. Later, look for a trace file with a modification date close to the finish time. If you're not sure of your finish time, look at files with modification dates later than your starting time. Hopefully, there won't be so many people using the trace facility simultaneously that this becomes a difficult task. If you have multiple trace files all created around the same time, you may have to look at the SQL statements inside each file to identify the one that you generated.

Enabling the SQL Trace feature

You can turn on the SQL Trace feature in three ways:

1. Issue an `ALTER SESSION` command.
2. Make a call to `DBMS_SYSTEM.SET_SQL_TRACE_IN_SESSION`.
3. Set `SQL_TRACE=TRUE` in the database parameter file.

The method that you choose depends on whether you want to enable tracing for your session, for someone else's session, or for all sessions connected to the database.

Enabling SQL Trace for Your Session

If you're logging on through SQL*Plus to test a few SQL statements, you can turn tracing on by using the `ALTER SESSION` commands. The following two commands turn tracing on and then off:

```
ALTER SESSION SET SQL_TRACE = TRUE;
ALTER SESSION SET SQL_TRACE = FALSE;
```

In between these two commands, you should execute the SQL queries that you are interested in tracing.

Enabling SQL Trace for Another Session

Most often, you will want to enable tracing for some other session besides your own. You may have a batch process that is taking a long time to run or an online program that is responding slowly. In that case, you will need to follow these steps to trace the SQL statements being executed:

1. Start the batch process or online program that you are interested in tracing.
2. Start another session using SQL*Plus.
3. Issue a `SELECT` statement against the `V$SESSION` view to determine the SID and serial number of the session created in step 1 by the program that you want to trace.
4. Issue a call to `DBMS_SYSTEM.SET_SQL_TRACE_IN_SESSION` to turn tracing on for that session.

5. Collect the information that you need.
6. Issue a call to DBMS_SYSTEM.SET_SQL_TRACE_IN_SESSION to turn tracing off.

The DBMS_SYSTEM.SET_SQL_TRACE_IN_SESSION procedure requires that you identify the specific session that you want to trace by supplying both the session identifier (SID) and the serial #. You can get this information from the V$SESSION view, which tells you who is logged on to the database. Here's an example of the query to use:

```
SQL> SELECT username, sid, serial#
  2  FROM v$session;

USERNAME                          SID   SERIAL#
------------------------------ --------- ---------
...
SEAPARK                             9       658
SYSTEM                             11      1134
```

Once you have the SID and serial # of the session that you want to trace, you can enable and disable tracing, as shown in this example:

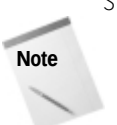
```
SQL> EXECUTE DBMS_SYSTEM.SET_SQL_TRACE_IN_SESSION (9,658,TRUE);

PL/SQL procedure successfully completed.

SQL> EXECUTE DBMS_SYSTEM.SET_SQL_TRACE_IN_SESSION (9,658,FALSE);

PL/SQL procedure successfully completed.
```

In this example, tracing was turned on for the SEAPARK user. The first call to DBMS_SYSTEM.SQL_TRACE_IN_SESSION included a value of TRUE to enable tracing. The second call included a value of FALSE to stop it.

Enabling SQL Trace for All Sessions

You can enable tracing for all sessions connecting to a database by placing the following entry in the database parameter file and then *bouncing* the database:

```
SQL_TRACE = TRUE
```



To *bounce* the database means to stop it and then restart it.

If you're running Oracle Parallel Server, this parameter will apply only to instances that read the parameter file that you change. If you bounce only one instance, then only sessions connected to that one instance will be traced.

Use care when you enable tracing on a database-wide basis like this. Some performance overhead is involved. Make sure that you really need database-wide statistics, and whatever you do, remember to turn it off later by setting SQL_TRACE = FALSE.

Using the TKPROF command

The raw trace files generated by Oracle aren't very readable. To view the results of a trace, you must run the TKPROF utility against the trace file that you generated. The TKPROF utility will read the trace file and create a new file containing the information in human readable form.

The TKPROF utility allows you to sort the queries in the output file based on a number of different parameters. For example, you can choose to have the queries that consumed the most CPU sorted to the front of the file. With large files, sorting the results makes it easier for you to quickly identify those queries most in need of work.

The TKPROF utility also provides you with the option of issuing an EXPLAIN PLAN for each statement in the trace file. The result here is that the output file will contain execution plans, in addition to the statistics, for each statement.

Using TKPROF Syntax

The TKPROF utility is a command-line utility. You run it from the command prompt, and you pass information to TKPROF using a series of command-line parameters. The syntax looks like this:

```
tkprof tracefile outputfile
       [explain=username/password]
       [table=[schema.]tablename]
       [print=integer]
       [aggregate={yes|no}]
       [insert=filename]
       [sys={yes|no}]
       [sort=option[,option...]]
```

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

- tracefile — Specifies the name of the trace file to be formatted.
- outputfile — Specifies the name of the output file that you want to create. This file will contain the formatted trace output.
- explain=*username*/*password* — Causes TKPROF to issue an EXPLAIN PLAN for each SQL statement in the trace file. To do this, TKPROF will connect using the username and password that you specify.

- `table=[schema.]tablename` — Specifies an alternate plan table to use. When you use the explain option, `TKPROF` expects to find a plan table named `PLAN_TABLE`. If your plan table is named differently, use this option.
- `print=integer` — Causes `TKPROF` to generate output only for the first *integer* SQL statements found in the trace file.
- `aggregate=yes|no` — Controls whether `TKPROF` reports multiple executions of the same SQL statement in summary form. The default is yes. If you say no, your output file could be quite large, as each execution of each statement will be listed separately.
- `insert=filename` — Causes `TKPROF` to generate a file containing `INSERT` statements. These `INSERT` statements will contain the trace information and can be used to save the trace data in a database table.
- `sys={yes|no}` — Controls whether recursive SQL statements and SQL statements executed by the user SYS are included in the output file. The default value is yes.
- `record=filename` — Creates a SQL script in the specified file that contains all the user-issued SQL statements in the trace file. You can use this file to recreate and replay the events that were traced.
- `sort=option[, option...]` — Sorts the trace output on the options that you specify.

Table 19-3 describes the available sort options.

Table 19-3
TKPROF Sort Options

Sort Option	*Description*
`prscnt`	The number of times a statement was parsed
`prscpu`	The amount of CPU time used for parsing
`prsela`	The elapsed parse time
`prsdsk`	The number of physical disk reads necessary during parsing
`prsqry`	The number of buffers accessed for a consistent read during parsing
`prscu`	The number of buffers accessed for a current read during parsing
`prsmis`	The number of library cache misses when a statement was parsed
`execnt`	The number of times the statement was executed
`execpu`	The amount of CPU time used for statement execution
`exeela`	The elapsed statement execution time

Sort Option	Description
exedsk	The number of physical disk reads necessary during execution
exeqry	The number of buffers accessed for a consistent read during execution
execu	The number of buffers accessed for a current read during execution
exerow	The number of rows processed during statement execution
exemis	The number of library cache misses during statement execution
fchcnt	The number of fetches used to return data for the statement
fchcpu	The amount of CPU time spent on fetches
fchela	The elapsed time spent on fetches
fchdsk	The number of physical disk reads made during a fetch
fchqry	The number of buffers accessed for a consistent read during a fetch
fchcu	The number of buffers accessed for a current read during a fetch
fchrow	The total number of rows fetched

If you're ever uncertain of the syntax, you can get a quick reminder by invoking TKPROF without passing any arguments. The TKPROF utility will display a help screen in response that contains a brief syntax summary.

Executing TKPROF

The simplest way to execute TKPROF is to specify an input and an output file name, like this:

```
$tkprof ora00242.trc ora00242.lst

TKPROF: Release 8.1.5.0.0 - Production on Tue Sep 14 09:56:03 1999

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

If your trace file contains a lot of SQL statements, you will want to sort the results. You'll likely find it beneficial to sort by the amount of CPU consumed when executing a statement. Consider this example:

```
$tkprof ora00242.trc ora00242.lst sort=execpu

TKPROF: Release 8.1.5.0.0 - Production on Tue Sep 14 09:57:41 1999

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

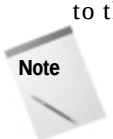
It's usually wise to generate execution plans for each SQL statement in the file. If you look at the execution plan for a poorly performing statement, it can help you better understand the problem.

Explaining Plans

TKPROF's `explain` option allows you to generate execution plans for the SQL statements in the trace file. When you use the explain option, you must supply `TKPROF` with a username and password. `TKPROF` uses it to log on and explain the plans. The user that you specify must have access to the tables referenced by the queries in the trace file. The user should also own a plan table named `PLAN_TABLE`. However, if no plan table currently exists, and if the user has the `CREATE TABLE` privilege, `TKPROF` will create one temporarily. Here's an example of the explain option being used:

```
$tkprof ora00242.trc ora00242.lst sort=execpu explain=seapark/seapark

TKPROF: Release 8.1.5.0.0 - Production on Tue Sep 14 10:05:33 1999

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

You'll notice that when you use the `explain` option, TKPROF's execution time is much longer than otherwise. This is due to the overhead involved in connecting to the database and issuing `EXPLAIN PLAN` statements for each query.



On UNIX systems, avoid passing a password on the command line. Any user issuing the `ps` command will be able to see it. Just pass in your username, and allow the utility that you are running to prompt you for your password.

One very important point about `TKPROF`'s `explain` option is that the plans are generated when you run `TKPROF`, not when the tracing is done. Consequently, the execution plan for a given statement may not reflect the plan that was actually used when the statement was executed. If you've created indexes in between the tracing and the running of `TKPROF`, or if you reanalyze the tables during that time, the plans that `TKPROF` generates may differ from those that were actually used when the trace was done.

Interpreting TKPROF's output

When you look at the file containing the `TKPROF` output, you'll see a section like the one shown in Listing 19-5 for each SQL statement.

Listing 19-5: TKPROF output for a SQL statement

```
SELECT a.id_no, a.animal_name, a.tank_no,
       NVL(c.caretaker_name, t.chief_caretaker_name)
FROM aquatic_animal a, tank t, caretaker c
WHERE a.tank_no = t.tank_no
AND c.caretaker_name (+) = t.chief_caretaker_name
ORDER BY a.tank_no, a.animal_name
```

```
call     count  cpu elapsed disk query current rows
------- ------ ---- ------- ---- ----- ------- ----
Parse        1 0.09    0.25    8   166       0    0
Execute      1 0.00    0.00    0     0       0    0
Fetch        2 0.00    0.03    7     6       8   10
------- ------ ---- ------- ---- ----- ------- ----
total        4 0.09    0.28   15   172       8   10

Misses in library cache during parse: 1
Optimizer goal: CHOOSE
Parsing user id: 57  (SEAPARK)

Rows Row Source Operation
---- -------------------------------------
  10 SORT ORDER BY
  10  HASH JOIN
   3   NESTED LOOPS OUTER
   4    TABLE ACCESS FULL TANK
   2    INDEX UNIQUE SCAN (object id 13669)
  10   TABLE ACCESS FULL AQUATIC_ANIMAL

Rows Execution Plan
---- -------------------------------------------------------
  0  SELECT STATEMENT   GOAL: CHOOSE
 10   SORT (ORDER BY)
 10    HASH JOIN
  3     NESTED LOOPS (OUTER)
  4      TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'TANK'
  2      INDEX   GOAL: ANALYZED (UNIQUE SCAN)
           OF 'CARETAKER_PK' (UNIQUE)
 10     TABLE ACCESS   GOAL: ANALYZED (FULL)
          OF 'AQUATIC_ANIMAL'
```

The major sections that you see here include the following:

- ✦ The SQL statement itself
- ✦ Statistics related to the execution of that SQL statement
- ✦ Information about the execution plan used for the statement

The statistics are often the key to diagnosing performance problems with specific SQL statements. To interpret this output, you need to understand what those statistics are telling you.

Understanding TKPROF's statistics

The first set of statistics that TKPROF displays for a statement is in tabular form. One row exists for each phase of SQL statement processing, plus a summary row at the bottom. The three phases of statement processing are:

- **The parse phase.** In this phase, Oracle takes a human-readable SQL statement and translates it into an execution plan that it can understand. This is where syntax is checked, object security is checked, and so forth.
- **The execution phase.** Most of the work occurs in this phase, especially for INSERT, UPDATE, and DELETE statements. With SELECT statements, the execution phase is where Oracle identifies all the rows that are to be returned. Any sorting, grouping, or summarizing takes place here.
- **The fetch phase.** This phase applies only to SELECT statements, and it is where Oracle sends the selected data to the application.

The columns that you see in the tabular statistics represent various counts and timings.

If the TIMED_STATISTICS parameter is not TRUE, you will see zeros for all the timings.

The column descriptions are as follows:

- COUNT — Tells you the number of times that a SQL statement was parsed or executed. For SELECT statements, this tells you the number of fetches that were made to retrieve the data.
- CPU — Tells you the amount of CPU time spent in each phase.
- ELAPSED — Tells you the elapsed time spent in each phase.
- DISK — Tells you the number of database blocks that were physically read from disk in each phase.
- QUERY — Tells you the number of buffers that were retrieved in consistent mode (usually for queries) in each phase.
- CURRENT — Tells you the number of buffers that were retrieved in current mode in each phase.

Following the tabular statistics, TKPROF reports the number of library cache misses, the optimizer goal, and the numeric user ID of the user who parsed the statement. The sidebar describes what to look for when running a trace.

Key Items to Look for When Running a Trace

When I run a trace, it's usually in response to someone's complaint about poor performance. So I'm looking for statements that take a long time to execute, especially relative to the number of rows that they return. With that in mind, here are some important items to watch for:

- A high CPU or elapsed time. If you are tuning a two-hour batch process, don't waste your time on statements that account for only a few seconds of that process. Focus your efforts on those statements that consistently consume the most time.
- A high number of disk reads relative to the number of reads in the query and current columns. This could indicate that tablescans are occurring or that your database buffer cache isn't large enough. If the disk reads exceed 10 percent of `query+disk+current`, then you may have cause for concern. However, if a query is doing a tablescan and you want it to do a tablescan, then a high percentage of disk reads is not such a concern.
- A high parse count. Ideally, a SQL statement should be parsed only once. If your parse counts are consistently higher, consider increasing the size of your shared pool. A high parse count could also indicate that your SQL statements are not using bind variables. Check your programs to see if they are building a new SQL statement for each execution.
- Library cache misses. A high number of library cache misses also indicates that your shared pool size is too small.

Once you've used the statistics to identify potentially troublesome statements, you can look at the execution plan to determine what is causing the problem.

Summary

In this chapter, you learned:

- You can use the SQL*Plus `EXPLAIN PLAN` statement to find out the execution plan for a SQL statement. To use `EXPLAIN PLAN`, you must create a plan table to hold the results. The Oracle-supplied `Utlxplan.sql` script will create this table for you, and you can find it in the `$ORACLE_HOME/rdbms/admin` directory.
- Optimizer hints allow you to tell Oracle how to execute a SQL statement. Hints are embedded in comments. Comments containing hints must immediately follow the keyword beginning the SQL statement and must take the form `/*+ ... */`. Use hints sparingly, and make sure that your table and index statistics are kept up-to-date.

- The `autotrace` option of SQL*Plus provides another way to get at the execution plan for a query. It's more convenient to use than `EXPLAIN PLAN` but carries with it the disadvantage of actually having to execute the statement being explained. If you are dealing with a statement that generates a lot of I/O or that takes a long time, use `EXPLAIN PLAN`, not `autotrace`.
- To help yourself identify poor performing SQL queries, you can use SQL Trace. SQL Trace is an Oracle utility that captures statistics about SQL statements while they are running. Later, you can review these statistics, identify problem statements, and take corrective action.
- When you use SQL Trace, the statistics that it captures are written to a trace file. You must use the `TKPROF` utility to process that trace file before you can view the results.

✦ ✦ ✦