XTREME SQL TUNING: THE TUNING LIMBO

Iggy Fernandez, Database Specialists

This paper is based on the chapter on SQL tuning in my book—*Beginning Oracle Database 11g Administration* (Apress, 2009). I present a fairly typical SQL statement and improve it in stages until it hits the theoretical maximum level of performance that is possible to achieve.

DEFINING EFFICIENCY

The efficiency of an SQL statement is measured by the amount of computing resources such as CPU cycles used in producing the output. Reducing the consumption of resources is the goal of SQL tuning. Elapsed time is not a good measure of the efficiency of an SQL statement because it is not always proportional to the amount of resources consumed. For example, contention for CPU cycles and disk I/O causes execution delays. The number of logical read operations is a better way to measure the consumption of computing resources because it is directly proportional to the amount of resources consumed—the fewer the number of logical read operations, the less CPU consumption.

TUNING BY EXAMPLE

Let's experience the process of tuning an SQL statement by creating two tables, `My_tables` and `My_indexes`, modeled after `dba_tables` and `dba_indexes`, two dictionary views. Every record in `My_tables` describes one table, and every record in `My_indexes` describes one index. The exercise is to print the details (`owner`, `table_name`, and `tablespace_name`) of tables that have at least one bitmap index.

Note that you will need the `SELECT ANY DICTIONARY` and `ALTER SYSTEM` privileges for this exercise. You may need the help of your database administrator or you may need to use your own Oracle database on your laptop or PC.

CREATING AND POPULATING THE TABLES

Create and populate the two tables as follows. When I performed the experiment, 1,574 rows of information were inserted into `My_tables`, and 1,924 rows of information were inserted into `My_indexes`.

```
SQL> /* Create the my_tables table */
SQL> CREATE TABLE my_tables AS
  2  SELECT dba_tables.*
  3    FROM dba_tables;

Table created.

SQL> /* Create the my_indexes table */
SQL> CREATE TABLE my_indexes AS
  2  SELECT dba_indexes.*
  3    FROM dba_tables, dba_indexes
  4   WHERE dba_tables.owner = dba_indexes.table_owner
  5     AND dba_tables.table_name = dba_indexes.table_name;

Table created.

SQL> /* Count the records in the my_tables table */
SQL> SELECT count(*)
  2    FROM my_tables;

  COUNT(*)
----------
      1574

SQL> /* Count the records in the my_indexes table */
SQL> SELECT count(*)
  2    FROM my_indexes;
```

```
  COUNT(*)
----------
      1924
```

Establishing a Baseline

Here is a simple SQL statement that prints the required information (`owner`, `table_name`, and `tablespace_name`) about tables that have an index of a specified type. We join `My_tables` and `My_indexes`, extract the required pieces of information from `My_tables`, and eliminate duplicates by using the `DISTINCT` operator:

```
SELECT DISTINCT my_tables.owner,
                my_tables.table_name,
                my_tables.tablespace_name
           FROM my_tables, my_indexes
          WHERE my_tables.owner = my_indexes.table_owner
            AND my_tables.table_name = my_indexes.table_name
            AND my_indexes.index_type = :index_type;
```

If you have some prior experience in SQL tuning, you will observe that we have not yet created appropriate indexes for `My_tables` and `My_indexes`, nor have we collected statistical information for use in constructing query execution plans. However, you will soon see that Oracle can execute queries and join tables even in the absence of indexes and can even generate statistical information dynamically by sampling the data.

Some preliminaries are required before we execute the query. We must activate the autotrace feature and must ensure that detailed timing information is captured during the course of the query (`statistics_level=ALL`). The autotrace feature is used to print the most important items of information relating to efficiency of SQL queries; more-detailed information is captured by Oracle if `statistics_level` is appropriately configured.

```
SQL> ALTER SESSION SET statistics_level=ALL;

Session altered.

SQL> SET AUTOTRACE on statistics
Here is the output of our first attempt:
SQL> /* First execution */
SQL> SELECT DISTINCT my_tables.owner,
  2                  my_tables.table_name,
  3                  my_tables.tablespace_name
  4             FROM my_tables, my_indexes
  5            WHERE my_tables.owner = my_indexes.table_owner
  6              AND my_tables.table_name = my_indexes.table_name
  7              AND my_indexes.index_type = :index_type;

OWNER                TABLE_NAME           TABLESPACE_NAME
-------------------- -------------------- --------------------
SH                   FWEEK_PSCAT_SALES_MV EXAMPLE
SH                   PRODUCTS             EXAMPLE
SH                   CUSTOMERS            EXAMPLE
SH                   SALES
SH                   COSTS


Statistics
----------------------------------------------------------
        114  recursive calls
          0  db block gets
        228  consistent gets
          0  physical reads
          0  redo size
        645  bytes sent via SQL*Net to client
        381  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          2  sorts (memory)
          0  sorts (disk)
          5  rows processed
```

Five rows of data are printed. The name of the tablespace is not printed in the case of two tables; the explanation is left as an exercise for you. But the more interesting information is that which relates the efficiency of the query; this has been printed because we activated the autotrace feature. Here is an explanation of the various items of information:

- *Recursive calls* result from all the work that is done behind the scenes by Oracle before it can begin executing your query. When a query is submitted for execution, Oracle first checks its syntax. It then checks the semantics—that is, it de-references synonyms and views and identifies the underlying tables. It then computes the *signature* (a.k.a. *hash value*) of the query and checks its cache of query execution plans for a reusable query plan that corresponds to that signature. If a query plan is not found, Oracle has to construct one; this results in *recursive calls*. The number of recursive calls required can be large if the information Oracle needs to construct the query plan is not available in the *dictionary cache* and has to be retrieved from the database or if there are no statistics about tables mentioned in the query and data blocks have to be sampled to generate statistics dynamically.
- *DB block gets* is the number of times Oracle needs the latest version of a data block. Oracle does not need the latest version of a data block when the data is simply being read by the user. Instead, Oracle needs the version of the data block that was current when the query started. The latest version of a data block is typically required when the intention is to modify the data.
- *Consistent gets*, a.k.a. *logical reads*, is the number of operations to retrieve a consistent version of a data block that were performed by Oracle while constructing query plans and while executing queries. Remember that all data blocks read during the execution of any query are the versions that were current when the query started; this is called read consistency. One of the principal objectives of query tuning is to reduce the number of consistent get operations that are required.
- *Physical reads* is the number of operations to read data blocks from the disks that were performed by Oracle because the blocks were not found in Oracle's cache when they were required. The limit on the number of physical reads is therefore the number of consistent gets.
- *Redo size* is the amount of information Oracle writes to the journals that track changes to the database. This metric applies only when the data is changed in some way.
- *Bytes sent via SQL*Net to client*, *bytes received via SQL*Net from client*, and *SQL*Net roundtrips to/from client* are fairly self-explanatory; they track the number and size of messages sent to and from Oracle and the user.
- *Sorts (memory)* and *sorts (disk)* track the number of sorting operations performed by Oracle during the course of our query. Sorting operations are performed in memory if possible; they spill onto the disks if the data does not fit into Oracle's memory buffers. It is desirable for sorting to be performed in memory because reading and writing data to and from the disks are expensive operations. The amount of memory available to database sessions for sorting operations is controlled by the value of `PGA_AGGREGATE_TARGET`.
- Finally, *rows processed* is the number of rows of information required by the user.

The preceding explanations should make it clear that a lot of overhead is involved in executing a query—for example, physical reads are required if data blocks are not found in Oracle's cache. There were no physical reads necessary when we executed our query the first time, indicating that all the required blocks were found in the cache. We can gauge the extent of overhead by flushing the *shared pool* (cache of query execution plans) and *buffer cache* (cache of data blocks) and executing the query again:

```
SQL> ALTER SYSTEM FLUSH SHARED_POOL;

System altered.

SQL> ALTER SYSTEM FLUSH BUFFER_CACHE;

System altered.
```

The numbers rise dramatically; 1,653 recursive calls and 498 consistent gets (including 136 physical reads) are required after the flush. The number of sort operations also jumps—from 0 to 37:

```
Statistics
----------------------------------------------------------
       1653  recursive calls
          0  db block gets
        498  consistent gets
        136  physical reads
```

```
          0  redo size
        645  bytes sent via SQL*Net to client
        381  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
         37  sorts (memory)
          0  sorts (disk)
          5  rows processed
```

Let's now flush only the buffer cache and execute the query again to quantify the precise amount of overhead work involved in constructing a query execution plan. Recursive calls and sorts fall from 1,653 to 0, while consistent gets fall from 498 to 108. We realize that the amount of overhead work dwarfs the actual work done by the query. The number of consistent gets is 108; it will not go down if we repeat the query again. The baseline number that we will attempt to reduce in this tuning exercise is therefore 108. Notice that the number of physical reads is less than the number of consistent gets even though we flushed the buffer cache before executing the query. The reason is that the consistent gets metric counts a block as many times as it is accessed during the course of a query, whereas the physical reads metric counts a block only when it is brought into the buffer cache from the disks. A block might be accessed several times during the course of the query but might have to be brought into the buffer cache only once:

```
Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
        108  consistent gets
        104  physical reads
          0  redo size
        645  bytes sent via SQL*Net to client
        381  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          5  rows processed
```

Executing the query yet one more time, we see that physical reads has fallen to zero because all data blocks are found in Oracle's cache:

```
Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
        108  consistent gets
          0  physical reads
          0  redo size
        645  bytes sent via SQL*Net to client
        381  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          5  rows processed
```

Table 17-1 summarizes our findings. The first time we executed the query, Oracle had to construct a query execution plan but found most of the required data in its cache. The second time we executed the query, Oracle had to obtain all the data required to construct a query execution plan from the disks, and all the data blocks required during the actual execution of our query also had to be obtained from the disks. The third time we executed the query, Oracle did not have to construct an execution plan, but all the data bocks required during the execution of our query had to be obtained from the disks. The fourth time we executed the query, all the required data blocks were found in Oracle's cache.

Table 17-1. Overhead Work Required to Construct a Query Execution Plan

Metric	First Execution	Second Execution	Third Execution	Query Plan Overhead	Fourth Execution
Recursive calls	114	1653	0	1653	0
Consistent gets	228	498	108	390	108
Physical reads	0	136	104	32	0

Sorts	2	37	0	37	0

The number of consistent gets will not go down if we repeat the query a fifth time; 108 is therefore the baseline number that we will try to reduce in this tuning exercise.

Examining the Query Plan

It's now time to examine the query plan that Oracle has been using. A simple way to do so is to use the `dbms_xplan.display_cursor` procedure as follows:

```
SQL> SELECT *
  2    FROM TABLE (DBMS_XPLAN.display_cursor (NULL, NULL, 'TYPICAL IOSTATS LAST'));

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------

SQL_ID  45b7a5rs8ynxa, child number 0
-------------------------------------
SELECT DISTINCT my_tables.owner,
                my_tables.table_name,
                my_tables.tablespace_name
           FROM my_tables,
                my_indexes
          WHERE my_tables.owner = my_indexes.table_owner
            AND my_tables.table_name = my_indexes.table_name
            AND my_indexes.index_type = :index_type

Plan hash value: 457052432

-------------------------------------------------
| Id  | Operation           | Name       | Starts |
-------------------------------------------------
|   1 |  HASH UNIQUE        |            |      1 |
|*  2 |   HASH JOIN         |            |      1 |
|*  3 |    TABLE ACCESS FULL| MY_INDEXES |      1 |
|   4 |    TABLE ACCESS FULL| MY_TABLES  |      1 |
-------------------------------------------------

-------------------------------------------------------------------------------------------------
| Id  | E-Rows |E-Bytes| Cost (%CPU)| E-Time   | A-Rows |   A-Time    | Buffers |
-------------------------------------------------------------------------------------------------
|   1 |      8 |   800 |    30   (7)| 00:00:01 |      5 |00:00:00.04 |     108 |
|*  2 |      8 |   800 |    29   (4)| 00:00:01 |     15 |00:00:00.04 |     108 |
|*  3 |     15 |   735 |    15   (0)| 00:00:01 |     15 |00:00:00.01 |      58 |
|   4 |   1574 | 80274 |    13   (0)| 00:00:01 |   1574 |00:00:00.01 |      50 |
-------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("MY_TABLES"."OWNER"="MY_INDEXES"."TABLE_OWNER" AND
              "MY_TABLES"."TABLE_NAME"="MY_INDEXES"."TABLE_NAME")
   3 - filter("MY_INDEXES"."INDEX_TYPE"=:INDEX_TYPE)

Note
-----
   - dynamic sampling used for this statement
```

The hash signature of the SQL statement is `45b7a5rs8ynxa` and that of the query execution plan is `457052432`. You may obtain different values when you try the exercise in your database. Oracle caches as many plans as possible for future reuse and knows how to translate a hash value into the precise place in memory where the SQL statement or plan is stored. Whenever an SQL statement is submitted for execution, Oracle computes its signature and searches for a statement with identical text and semantics in the cache. If a statement with identical text and semantics is found, its plan can be reused. If Oracle finds a statement with identical text but does not find one that also has identical semantics, it must create a new execution plan; this execution plan is called a *child cursor*. Multiple plans can exist for the same statement—this is indicated by

the *child number*. For example, two users may each own a private copy of a table mentioned in an SQL statement, and therefore different query plans will be needed for each user.

The execution plan itself is shown in tabular format and is followed by the list of filters (a.k.a. *predicates*) that apply to the rows of the table. Each row in the table represents one step of the query execution plan. Here is the description of the columns:

- *Id* is simply the row number.
- *Operation* describes the step; for example, `TABLE ACCESS FULL` means that all rows in the table are retrieved.
- *Name* is an optional piece of information; it is the name of the object to which the step applies.
- *Starts* is the number of times the step is executed.
- *E-Rows* is the number of rows that Oracle expected to obtain in the step.
- *E-Bytes* is the total number of bytes that Oracle expected to obtain in the step.
- *Cost* is the cumulative cost that Oracle expected to incur at the end of the step. The unit is block equivalents—that is, the number of data blocks that could be retrieved in the same amount of time. The percentage of time for which Oracle expects to use the CPU is displayed along with the cost.
- *E-time* is an estimate of the cumulative amount of time to complete this step and all prior steps. Hours, minutes, and seconds are displayed.
- *A-Rows* is the number of rows that Oracle actually obtained in the step.
- *A-Bytes* is the total number of bytes that Oracle actually obtained in the step.
- *Buffers* is the cumulative number of consistent get operations that were actually performed in the step and all prior steps.

More than a little skill is needed to interpret the execution plan. Notice the indentation of the operations listed in the second column. The correct sequence of operations is obtained by applying the following rule: Perform operations in the order in which they are listed except that if the operations listed after a certain operation are more deeply indented in the listing, then perform those operations first (in the order in which those operations are listed). On applying this rule, we see that the operations are performed in the following order:

```
-----------------------------------------
| Id  | Operation           | Name        |
-----------------------------------------
|*  3 |    TABLE ACCESS FULL| MY_INDEXES  |
|   4 |    TABLE ACCESS FULL| MY_TABLES   |
|*  2 |   HASH JOIN         |             |
|   1 |  HASH UNIQUE        |             |
-----------------------------------------
```

First, all the rows of data in the `My_indexes` table will be retrieved (`TABLE ACCESS FULL`), and a lookup table (a.k.a. *hash table*) is constructed from rows that satisfy the filter (a.k.a. *predicate*) `my_indexes.index_type = :index_type`. This means that a signature (a.k.a. *hash value*) is calculated for each row of data satisfying the predicate. This signature is then translated into a position in the hash table where the row will be stored. This makes it tremendously easy to locate the row when it is required later. Because `My_indexes` and `My_tables` are related by the values of `owner` and `table_name`, a hash signature will be constructed from these values.

Next, all the rows of data in the My_tables table will be retrieved (`TABLE ACCESS FULL`). After retrieving each row of data from the `My_tables` table, rows of data from the `My_indexes` table satisfying the join predicate `my_tables.owner = my_indexes.owner AND my_tables.table_name = my_indexes.table_name` will be retrieved from the lookup table constructed in the previous step (`HASH JOIN`) and, if any such rows are found, the required data items (`owner, table_name, and tablespace_name`) are included in the query results. Duplicates are avoided (`HASH UNIQUE`) by storing the resulting rows of data in another hash table; new rows are added to the hash table only if a matching row is not found.

Of particular interest are the columns in the execution plan labeled *E-Rows*, *E-Bytes*, *Cost*, and *E-Time*; these are estimates generated by Oracle. Time and cost are straightforward computations based on rows and bytes; but how does Oracle estimate rows and bytes in the first place? The answer is that Oracle uses a combination of rules of thumb and any available statistical information available in the data dictionary; this statistical information is automatically refreshed every night. In the absence of

statistical information, Oracle samples a certain number of data blocks from the table and estimates the number of qualifying rows in each block of the table and the average length of these rows. This is indicated by the remark `dynamic sampling used for this statement` seen at the end of the preceding listing.

In our example, Oracle estimated that 15 qualifying rows would be found in the `My_indexes` table and that 8 rows would remain after the qualifying rows were matched to the rows in the `My_tables` table. Oracle's estimate of the number of qualifying rows in `My_indexes` is surprisingly accurate, but its estimate of the number of rows that could be matched to rows in the `My_tables` table is less accurate. Every row in `My_indexes` corresponds to exactly one row in the `My_tables` table and, therefore, a better estimate would have been 15. The reason Oracle cannot estimate correctly in this case is that it is unaware of the relationship between the tables—we have not yet given Oracle the information it needs. Oracle therefore resorts to a rule of thumb; it estimates that each row in a table will match the same number of rows in the table to which it is joined—the number obtained by dividing the number of rows (or estimate thereof) in the table to which the row is being joined by the number of distinct values (or estimate thereof) in that table. (Another equally plausible estimate can be made by reversing the roles of the two tables involved in the computation, and Oracle chooses the lower of the two estimates.)

INDEXES AND STATISTICS

Let's create an appropriate set of indexes on our tables. First, we define primary key constraints on `My_tables` and `My_indexes`. Oracle automatically creates unique indexes because it needs an efficient method of enforcing the constraints. We also create a foreign key constraint linking the two tables and create an index on the foreign key. Finally, we create an index on the values of `index_type` because our query restricts the value of `index_type`. We then collect statistics on both tables as well as the newly created indexes as follows:

```
ALTER TABLE my_tables
ADD (CONSTRAINT my_tables_pk PRIMARY KEY (owner, table_name));

ALTER TABLE my_indexes
ADD (CONSTRAINT my_indexes_pk PRIMARY KEY (owner, index_name));

ALTER TABLE my_indexes
ADD (CONSTRAINT my_indexes_fk1 FOREIGN KEY (table_owner, table_name)
REFERENCES my_tables);

CREATE INDEX my_indexes_i1 ON my_indexes (index_type);

CREATE INDEX my_indexes_fk1 ON my_indexes (table_owner, table_name);

EXEC DBMS_STATS.gather_table_stats(ownname=>'IFERNANDEZ',tabname=>'MY_TABLES');
EXEC DBMS_STATS.gather_table_stats(ownname=>'IFERNANDEZ',tabname=>'MY_INDEXES');
EXEC DBMS_STATS.gather_index_stats(ownname=>'IFERNANDEZ',indname=>'MY_TABLES_PK');
EXEC DBMS_STATS.gather_index_stats(ownname=>'IFERNANDEZ',indname=>'MY_INDEXES_I1');
EXEC DBMS_STATS.gather_index_stats(ownname=>'IFERNANDEZ',indname=>'MY_INDEXES_FK1');
```

We find a substantial improvement both in the accuracy of the estimates and in the efficiency of the query plan after indexes are created and statistics are gathered. Oracle now correctly estimates that every qualifying row in the `My_indexes` table will match exactly one row in the `My_tables` table, and the remark `dynamic sampling used for this statement` is not printed. Oracle uses the index on the values of `index_type` to efficiently retrieve the 15 qualifying rows—this reduces the number of consistent gets from 108 to 55.

```
SQL> SELECT *
  2    FROM TABLE (DBMS_XPLAN.display_cursor (NULL, NULL, 'TYPICAL IOSTATS LAST'));

PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------

SQL_ID  45b7a5rs8ynxa, child number 0
-------------------------------------
SELECT DISTINCT my_tables.owner,
                my_tables.table_name,
                my_tables.tablespace_name
           FROM my_tables,
                my_indexes
          WHERE my_tables.owner = my_indexes.table_owner
```

```
            AND my_tables.table_name = my_indexes.table_name
            AND my_indexes.index_type = :index_type

Plan hash value: 1434988034

-------------------------------------------------------------------
| Id  | Operation                     | Name          | Starts |
-------------------------------------------------------------------
|   1 |  HASH UNIQUE                  |               |      1 |
|*  2 |   HASH JOIN                   |               |      1 |
|   3 |    TABLE ACCESS BY INDEX ROWID| MY_INDEXES    |      1 |
|*  4 |     INDEX RANGE SCAN          | MY_INDEXES_I1 |      1 |
|   5 |    TABLE ACCESS FULL          | MY_TABLES     |      1 |
-------------------------------------------------------------------

-------------------------------------------------------------------------------------------
| Id  | E-Rows |E-Bytes| Cost (%CPU)| E-Time   | A-Rows |   A-Time    | Buffers |
-------------------------------------------------------------------------------------------
|   1 |     15 |   945 |     17  (12)| 00:00:01 |      5 |00:00:00.02 |      55 |
|*  2 |     15 |   945 |     16   (7)| 00:00:01 |     15 |00:00:00.02 |      55 |
|   3 |     15 |   465 |      2   (0)| 00:00:01 |     15 |00:00:00.01 |       5 |
|*  4 |     15 |       |      1   (0)| 00:00:01 |     15 |00:00:00.01 |       2 |
|   5 |   1574 | 50368 |     13   (0)| 00:00:01 |   1574 |00:00:00.01 |      50 |
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("MY_TABLES"."OWNER"="MY_INDEXES"."TABLE_OWNER" AND
              "MY_TABLES"."TABLE_NAME"="MY_INDEXES"."TABLE_NAME")
   4 - access("MY_INDEXES"."INDEX_TYPE"=:INDEX_TYPE)
```

Using SQL Access Advisor

If you are using Enterprise Edition and have a license for the Tuning Pack, you can use SQL Tuning Advisor to help you with SQL tuning. We've already created indexes and generated statistics; we now look to SQL Tuning Advisor for fresh ideas. The commands in the following example generate a recommendation report:

```
SQL> /* get recommendations from SQL Access Advisor */
SQL> VARIABLE tuning_task VARCHAR2(32);
SQL> EXEC :tuning_task := dbms_sqltune.create_tuning_task (sql_id => '&sqlID');
Enter value for sqlid: 45b7a5rs8ynxa

PL/SQL procedure successfully completed.

SQL>
SQL> EXEC dbms_sqltune.execute_tuning_task(task_name => :tuning_task);

PL/SQL procedure successfully completed.

SQL>
SQL> SET LONG 100000;
SQL> SET PAGESIZE 1000
SQL> SET LINESIZE 160
SQL> COLUMN recommendations FORMAT a160
SQL>
SQL> SELECT DBMS_SQLTUNE.report_tuning_task (:tuning_task) AS recommendations
  2    FROM DUAL;
```

The final SELECT in the preceding code retrieves a report from the SQL Tuning Advisor. Following is that report:

```
RECOMMENDATIONS
--------------------------------------------------------------------------------

GENERAL INFORMATION SECTION
-------------------------------------------------------------------------------
Tuning Task Name                  : TASK_21
```

```
Tuning Task Owner                  : IFERNANDEZ
Scope                              : COMPREHENSIVE
Time Limit(seconds)                : 1800
Completion Status                  : COMPLETED
Started at                         : 09/30/2008 05:37:57
Completed at                       : 09/30/2008 05:37:58
Number of Index Findings           : 1

-------------------------------------------------------------------------------
Schema Name: IFERNANDEZ
SQL ID     : 45b7a5rs8ynxa
SQL Text   : SELECT DISTINCT my_tables.owner,
                             my_tables.table_name,
                             my_tables.tablespace_name
                        FROM my_tables, my_indexes
                       WHERE my_tables.owner = my_indexes.table_owner
                         AND my_tables.table_name = my_indexes.table_name
                         AND my_indexes.index_type = :index_type

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

1- Index Finding (see explain plans section below)
--------------------------------------------------
  The execution plan of this statement can be improved by creating one or more
  indices.

  Recommendation (estimated benefit: 100%)
  ----------------------------------------
  - Consider running the Access Advisor to improve the physical schema design
    or creating the recommended index.
    create index IFERNANDEZ.IDX$$_00150001 on
    IFERNANDEZ.MY_TABLES('OWNER','TABLE_NAME','TABLESPACE_NAME');

  Rationale
  ---------
    Creating the recommended indices significantly improves the execution plan
    of this statement. However, it might be preferable to run "Access Advisor"
    using a representative SQL workload as opposed to a single statement. This
    will allow you to get comprehensive index recommendations, which takes into
    account index maintenance overhead and additional space consumption.

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

1- Original
-----------
Plan hash value: 1434988034

-------------------------------------------------------------------------------------------
| Id |Operation                      |Name          |Rows|Bytes|Cost (%CPU)|Time    |
-------------------------------------------------------------------------------------------
|   0|SELECT STATEMENT               |              |  15|  945|   17  (12)|00:00:01|
|   1| HASH UNIQUE                   |              |  15|  945|   17  (12)|00:00:01|
|*  2|  HASH JOIN                    |              |  15|  945|   16   (7)|00:00:01|
|   3|   TABLE ACCESS BY INDEX ROWID|MY_INDEXES     |  15|  465|    2   (0)|00:00:01|
|*  4|    INDEX RANGE SCAN           |MY_INDEXES_I1  |  15|     |    1   (0)|00:00:01|
|   5|   TABLE ACCESS FULL           |MY_TABLES      |1574|50368|   13   (0)|00:00:01|
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
```

```
   2 - access("MY_TABLES"."OWNER"="MY_INDEXES"."TABLE_OWNER" AND
              "MY_TABLES"."TABLE_NAME"="MY_INDEXES"."TABLE_NAME")
   4 - access("MY_INDEXES"."INDEX_TYPE"=:INDEX_TYPE)

2- Using New Indices
--------------------
Plan hash value: 317787978

-------------------------------------------------------------------------------------------
| Id |Operation                      |Name          |Rows|Bytes|Cost (%CPU)|Time     |
-------------------------------------------------------------------------------------------
|   0|SELECT STATEMENT               |              |  15|  945|     8  (25)|00:00:01|
|   1| HASH UNIQUE                   |              |  15|  945|     8  (25)|00:00:01|
|*  2|  HASH JOIN                    |              |  15|  945|     7  (15)|00:00:01|
|   3|   TABLE ACCESS BY INDEX ROWID|MY_INDEXES     |  15|  465|     2   (0)|00:00:01|
|*  4|    INDEX RANGE SCAN           |MY_INDEXES_I1  |  15|     |     1   (0)|00:00:01|
|   5|   INDEX FAST FULL SCAN        |IDX$$_00150001|1574|50368|     4   (0)|00:00:01|
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("MY_TABLES"."OWNER"="MY_INDEXES"."TABLE_OWNER" AND
              "MY_TABLES"."TABLE_NAME"="MY_INDEXES"."TABLE_NAME")
   4 - access("MY_INDEXES"."INDEX_TYPE"=:INDEX_TYPE)

-------------------------------------------------------------------------------
```

SQL Tuning Advisor recommends that we create yet another index on the `My_tables` table. The new index would contain all the items of information required by our query and make it necessary to read the table itself. However, let's first make sure that Oracle is taking advantage of the indexes we have already created before creating yet another index. In particular, let's check whether the query can be more efficiently answered with the help of the foreign key index `my_indexes_fk1`.

Optimizer Hints

As I have pointed out before, hints for the optimizer can be embedded inside an SQL statement if the optimizer does not find an acceptable query plan for the statement. Each hint partially constrains the optimizer, and a full set of hints completely constrains the optimizer.[1] Let's instruct Oracle to use the `my_indexes_fk1` index to efficiently associate qualifying records in the `My_indexes` table with matching records from the `My_tables` table:

```
SELECT           /*+ INDEX(MY_INDEXES (INDEX_TYPE))
                     INDEX(MY_TABLES (OWNER TABLE_NAME))
                     LEADING(MY_INDEXES MY_TABLES)
                     USE_NL(MY_TABLES)
                 */
       DISTINCT my_tables.owner,
                my_tables.table_name,
                my_tables.tablespace_name
           FROM my_tables, my_indexes
          WHERE my_tables.owner = my_indexes.table_owner
            AND my_tables.table_name = my_indexes.table_name
            AND my_indexes.index_type = :index_type;
```

The `LEADING` hint specifies the order in which tables are processed, the `INDEX` hint specifies that an index be used, and the `USE_NL` hint specifies that tables be joined using the simple *nested loop* method (instead of the *hash* method used in previous executions). The use of the `my_indexes_fk1` index reduces the number of consistent gets to 37.

[1] The use of hints is a defense against *bind variable peeking*. To understand the problem, remember that a query execution plan is constructed once and used many times. If the values of the bind variables in the first invocation of the query are not particularly representative of the data, the query plan that is generated will not be a good candidate for reuse. *Stored outlines* are another defense against bind variable peeking. It is also possible to turn off bind variable peeking for your session or for the entire database; this forces Oracle to generate a query plan that is not tailored to one set of bind variables. However, note that bind variable peeking does work well when the data has a uniform distribution.

```
SQL> SELECT *
  2    FROM TABLE (DBMS_XPLAN.display_cursor (NULL, NULL, 'TYPICAL IOSTATS LAST'));

PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------

SQL_ID  dsh3tvd5p501p, child number 0
-------------------------------------
SELECT          /*+ INDEX(MY_INDEXES (INDEX_TYPE))
                    INDEX(MY_TABLES (OWNER TABLE_NAME))
                    LEADING(MY_INDEXES MY_TABLES)
                    USE_NL(MY_TABLES) */
       DISTINCT my_tables.owner,
                my_tables.table_name,
                my_tables.tablespace_name
           FROM my_tables,
                my_indexes
          WHERE my_tables.owner = my_indexes.table_owner
            AND my_tables.table_name = my_indexes.table_name
            AND my_indexes.index_type = :index_type

Plan hash value: 813127232

-----------------------------------------------------------------
| Id  | Operation                     | Name          | Starts |
-----------------------------------------------------------------
|   1 |  HASH UNIQUE                  |               |      1 |
|   2 |   NESTED LOOPS                |               |      1 |
|   3 |    TABLE ACCESS BY INDEX ROWID| MY_INDEXES    |      1 |
|*  4 |     INDEX RANGE SCAN          | MY_INDEXES_I1 |      1 |
|   5 |    TABLE ACCESS BY INDEX ROWID| MY_TABLES     |     15 |
|*  6 |     INDEX UNIQUE SCAN         | MY_TABLES_PK  |     15 |
-----------------------------------------------------------------

-------------------------------------------------------------------------------------------
| Id  | E-Rows |E-Bytes| Cost (%CPU)| E-Time   | A-Rows |   A-Time    | Buffers |
-------------------------------------------------------------------------------------------
|   1 |     15 |   945 |     18   (6)| 00:00:01 |      5 |00:00:00.01 |      37 |
|   2 |     15 |   945 |     17   (0)| 00:00:01 |     15 |00:00:00.01 |      37 |
|   3 |     15 |   465 |      2   (0)| 00:00:01 |     15 |00:00:00.01 |       5 |
|*  4 |     15 |       |      1   (0)| 00:00:01 |     15 |00:00:00.01 |       2 |
|   5 |      1 |    32 |      1   (0)| 00:00:01 |     15 |00:00:00.01 |      32 |
|*  6 |      1 |       |      0   (0)|          |     15 |00:00:00.01 |      17 |
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   4 - access("MY_INDEXES"."INDEX_TYPE"=:INDEX_TYPE)
   6 - access("MY_TABLES"."OWNER"="MY_INDEXES"."TABLE_OWNER" AND
              "MY_TABLES"."TABLE_NAME"="MY_INDEXES"."TABLE_NAME")
```

Extreme Tuning

It does not appear possible to reduce consistent get operations any further because we are already using indexes to maximum advantage. Here is a summary of the previous execution strategy:

- Using the `my_indexes_i1` index, collect the addresses of rows of the `My_indexes` table that qualify for inclusion; this requires at least one consistent get operation.
- Using the addresses obtained in the previous step, retrieve the qualifying rows from the `My_indexes` table. There are 15 rows that qualify and, therefore, we may have to visit 15 different blocks and perform 15 consistent get operations; fewer operations may be required if multiple rows are found when a block is visited.

- For every qualifying row retrieved from the `My_indexes` table, use the `my_tables_pk` index and obtain the address of the matching row in the `My_tables` table; this requires another 15 consistent get operations.
- Using the addresses obtained in the previous step, retrieve the matching rows from the `My_tables` table; this requires another 15 consistent get operations.
- Finally, eliminate duplicates from the result. This does not require any consistent get operations.

If we add up the numbers in this analysis, we get 46. The actual number of consistent get operations may be slightly higher or slightly lower than 46 if multiple rows are found when a block is visited or multiple blocks have to be visited to retrieve a row of information from a table or from an index. It therefore appears that we have tuned the SQL statement as much as we can. After all, we need to use indexes to retrieve records quickly, and we cannot avoid visiting two tables.

However, there is a way to retrieve records quickly without the use of indexes, and there is a way to retrieve data from two tables without actually visiting two tables. *Hash clusters* allow data to be retrieved without the use of indexes; Oracle computes the hash signature of the record you need and translates the hash signature into the address of the record. Clustering effects can also significantly reduce the number of blocks that need to be visited to retrieve the required data. Further, *materialized views* combine the data from multiple tables and enable you to retrieve data from multiple tables without actually visiting the tables themselves. We can combine hash clusters and materialized views to achieve a dramatic reduction in the number of consistent get operations; in fact we will only need one consistent get operation.

First, we create a hash cluster and then we combine the data from `My_tables` and `My_indexes` into another table called `My_tables_and_indexes`. In the interests of brevity, we select only those items of information required by our query; items of information that would help other queries would normally have been considered for inclusion:

```
SQL> CREATE CLUSTER my_cluster (index_type VARCHAR2(27))
  2  SIZE 8192 HASHKEYS 5;

Cluster created.
```

Next, we create a materialized view of the data. We use *materialized view logs* to ensure that any future changes to the data in `My_tables` or `My_indexes` are immediately made to the copy of the data in the materialized view. We enable *query rewrite* to ensure that our query is automatically modified and that the required data is retrieved from the materialized view instead of from `My_tables` and `My_indexes`. Appropriate indexes that would improve the usability and maintainability of the materialized view should also be created; I leave this as an exercise for you.

```
SQL> CREATE MATERIALIZED VIEW LOG ON my_tables WITH ROWID;

Materialized view log created.

SQL> CREATE MATERIALIZED VIEW LOG ON my_indexes WITH ROWID;

Materialized view log created.

SQL> CREATE MATERIALIZED VIEW my_mv
  2  CLUSTER my_cluster (index_type)
  3  REFRESH FAST ON COMMIT
  4  ENABLE QUERY REWRITE
  5  AS
  6  SELECT t.ROWID AS table_rowid,
  7         t.owner AS table_owner,
  8         t.table_name,
  9         t.tablespace_name,
 10         i.ROWID AS index_rowid,
 11         i.index_type
 12    FROM my_tables t,
 13         my_indexes i
 14   WHERE t.owner = i.table_owner
 15     AND t.table_name = i.table_name;

Materialized view created.

SQL> EXEC DBMS_STATS.gather_table_stats(ownname=>'IFERNANDEZ',tabname=>'MY_MV');
```

```
PL/SQL procedure successfully completed.
```

It's now time to test our query again; we find that the number of consistent gets has been reduced to its theoretical limit of 1:

```
SQL> SELECT *
  2    FROM TABLE (DBMS_XPLAN.display_cursor (NULL, NULL, 'TYPICAL IOSTATS LAST'));

PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------

SQL_ID  45b7a5rs8ynxa, child number 0
-------------------------------------
SELECT DISTINCT my_tables.owner,
                my_tables.table_name,
                my_tables.tablespace_name
           FROM my_tables,
                my_indexes
          WHERE my_tables.owner = my_indexes.table_owner
            AND my_tables.table_name = my_indexes.table_name
            AND my_indexes.index_type = :index_type

Plan hash value: 1006555447

-------------------------------------------
| Id  | Operation          | Name  | Starts |
-------------------------------------------
|   1 |  HASH UNIQUE       |       |      1 |
|*  2 |   TABLE ACCESS HASH| MY_MV |      1 |
-------------------------------------------

-----------------------------------------------------------------------------------------
| Id  | E-Rows |E-Bytes| Cost (%CPU)| E-Time   | A-Rows |   A-Time    | Buffers |
-----------------------------------------------------------------------------------------
|   1 |    321 | 12198 |     1 (100)| 00:00:01 |      5 |00:00:00.01 |       1 |
|*  2 |    321 | 12198 |            |          |     15 |00:00:00.01 |       1 |
-----------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("MY_MV"."INDEX_TYPE"=:INDEX_TYPE)
```

But Wait, There's More!

Executing a query requires at least one consistent get operation. However, there is no need to execute the query if the query results have been previously cached and the data has not changed. Oracle 11*g* introduced the `RESULT_CACHE` hint to indicate that the data should be obtained from cache if possible. Note that this feature is available only with the Enterprise Edition. In the following example, you can see that the number of consistent gets has been reduced to zero!

```
SQL> SELECT          /*+ RESULT_CACHE */
  2         DISTINCT my_tables.owner,
  3                  my_tables.table_name,
  4                  my_tables.tablespace_name
  5             FROM my_tables, my_indexes
  6            WHERE my_tables.owner = my_indexes.table_owner
  7              AND my_tables.table_name = my_indexes.table_name
  8              AND my_indexes.index_type = :index_type;

OWNER                TABLE_NAME           TABLESPACE_NAME
-------------------- -------------------- --------------------
SH                   FWEEK_PSCAT_SALES_MV EXAMPLE
SH                   PRODUCTS             EXAMPLE
SH                   CUSTOMERS            EXAMPLE
SH                   SALES
SH                   COSTS
```

```
Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
          0  consistent gets
          0  physical reads
          0  redo size
        652  bytes sent via SQL*Net to client
        416  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          5  rows processed

SQL> SELECT *
  2    FROM TABLE (DBMS_XPLAN.display_cursor (NULL, NULL, 'TYPICAL IOSTATS LAST'));

PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------

SQL_ID  cwht0anw8vv4s, child number 0
-------------------------------------
SELECT          /*+ RESULT_CACHE */
       DISTINCT my_tables.owner,
                my_tables.table_name,
                my_tables.tablespace_name
           FROM my_tables,
                my_indexes
          WHERE my_tables.owner = my_indexes.table_owner
            AND my_tables.table_name = my_indexes.table_name
            AND my_indexes.index_type = :index_type

Plan hash value: 1006555447

-----------------------------------------------------------------------
| Id  | Operation           | Name                          | Starts |
-----------------------------------------------------------------------
|   0 | SELECT STATEMENT    |                               |      1 |
|   1 |  RESULT CACHE       | afscr8p240b168b5az0dkd4k65    |      1 |
|   2 |   HASH UNIQUE       |                               |      0 |
|*  3 |    TABLE ACCESS HASH| MY_MV                         |      0 |
-----------------------------------------------------------------------

-------------------------------------------------------------------------------
| Id  | E-Rows |E-Bytes| Cost (%CPU)| E-Time   | A-Rows |   A-Time   |
-------------------------------------------------------------------------------
|   0 |        |       |     1 (100)|          |      5 |00:00:00.01 |
|   1 |        |       |            |          |      5 |00:00:00.01 |
|   2 |    602 | 24682 |     1 (100)| 00:00:01 |      0 |00:00:00.01 |
|*  3 |    602 | 24682 |            |          |      0 |00:00:00.01 |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - access("MY_MV"."INDEX_TYPE"=:INDEX_TYPE)

Result Cache Information (identified by operation id):
------------------------------------------------------

   1 -
```

SUMMARY

You saw four query plans in this exercise:

- The first query plan did not use any indexes because none had been created at the time.
- Qualifying records in `My_indexes` were efficiently identified by using the index on the values of `index_type` and were placed in a lookup table. All the records in `My_tables` were then retrieved and compared with the records in the lookup table.
- Qualifying records in `My_indexes` were efficiently identified using the index on the values of `index_type`. An index on the values of `table_owner` and `table_name` was then used to efficiently find the corresponding records in `My_tables`.
- A *materialized view* was used to combine the data from `My_tables` and `My_indexes`, and a *hash cluster* was used to create clusters of related records.

ABOUT THE AUTHOR

Iggy Fernandez is a senior staff consultant at Database Specialists and edits the technical journal of the Northern California Oracle Users Group (NoCOUG). He is the author of *Beginning Oracle Database 11*g *Administration* (Apress, 2009). His e-mail address is **iggy_fernandez@hotmail.com**.