Interpreting Query Execution Plans

Iggy Fernandez, Database Specialists

Introduction

SQL efficiency is central to database efficiency, and the ability to interpret SQL query execution plans is a critical skill of the application developer and database administrator. In this paper, I review the process of displaying and interpreting query execution plans. I also discuss how to generate graphical versions of query plans that are much easier to read than their more common tabular counterparts.

Much information about query plans is available in the literature and I will simply provide links to it instead of repeating it here.

What Is a Query Execution Plan?

SQL is a declarative language, not a procedural language. This means that a SQL query only specifies what data to retrieve, not how to retrieve it. A database component called the query optimizer decides how best to retrieve the data. For example, it decides the order in which tables are processed, how to join the tables (nested loops, hash join, merge join, etc), and which indexes to use if any.

EXPLAIN PLAN May Not Tell The Right Story

The traditional—and least effective—way of obtaining a query plan is to use `EXPLAIN PLAN`. Third-party pools such as Toad use this method behind the scenes. Prior to Oracle Database 10*g* , the plan had to be formatted manually using `CONNECT BY`. Beginning with Oracle Database 10*g*, you can use `DBMS_XPLAN` to format the plan. This method has the incurable defect that it may not produce the same plan that is actually used when the query is actually executed. The reasons are well documented in the literature and I will not repeat them here. For example, refer to the demonstration by Kerry Osborne. Furthermore, this method can only show Oracle's estimates of cardinalities and execution times for each step in the plan but not actual cardinalities and execution times. I only mention this method for completeness but do not recommend this method except as a teaching tool for beginners.

```
EXPLAIN PLAN SET statement_id = 'TEST' FOR
-- Employees who earn more than their managers
SELECT
  e1.employee_id,
  e1.salary,
  e2.employee_id,
  e2.salary
FROM
  employees e1,
  employees e2
WHERE
  e1.department_id = :department_id
  AND e1.manager_id = e2.employee_id
  AND e1.salary > e2.salary;

COLUMN operation FORMAT a40

SELECT
  cardinality,
  cost,
  lpad(' ',level-1)||operation||' '|| options||' '||object_name AS operation
FROM plan_table
```

```
CONNECT BY prior id = parent_id AND prior statement_id = statement_id
START WITH id = 0 AND statement_id = 'TEST'
ORDER BY id;

CARDINALITY       COST OPERATION
----------- ---------- ----------------------------------------
          1          6 SELECT STATEMENT
          1          6  HASH JOIN
         10          2   TABLE ACCESS BY INDEX ROWID EMPLOYEES
         10          1    INDEX RANGE SCAN EMP_DEPARTMENT_IX
        107          3   TABLE ACCESS FULL EMPLOYEES

SELECT * FROM TABLE(dbms_xplan.display);

PLAN_TABLE_OUTPUT
-------------------------------------------------------------------------------------------
Plan hash value: 2254211361

-----------------------------------------------------------------------------------------------
| Id  | Operation                    | Name             | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                  |     1 |    23 |     6  (17)| 00:00:01 |
|*  1 |  HASH JOIN                   |                  |     1 |    23 |     6  (17)| 00:00:01 |
|   2 |   TABLE ACCESS BY INDEX ROWID| EMPLOYEES        |    10 |   150 |     2   (0)| 00:00:01 |
|*  3 |    INDEX RANGE SCAN          | EMP_DEPARTMENT_IX |    10 |       |     1   (0)| 00:00:01 |
|   4 |   TABLE ACCESS FULL          | EMPLOYEES        |   107 |   856 |     3   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------

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

   1 - access("E1"."MANAGER_ID"="E2"."EMPLOYEE_ID")
       filter("E1"."SALARY">"E2"."SALARY")
   3 - access("E1"."DEPARTMENT_ID"=TO_NUMBER(:DEPARTMENT_ID))

18 rows selected.
```

Autotrace May Not Tell The Right Story

The second method of obtaining a query plan is to use `autotrace` in SQL*Plus. This method requires the use of the PLUSTRACE role and causes a plan to be displayed right after a query is executed in SQL*Plus. However it is not well known that this method uses `EXPLAIN PLAN` behind the scenes and can therefore show a different plan that the one which was actually used during the execution of the query; this can be terribly misleading. For example, refer to the demonstration by Kelly Osborne. Furthermore, it does not show the actual cardinalities and execution times for each step even though the query was actually executed. I do not recommend this method either and only mention it for completeness.

```
SET autotrace on

VARIABLE department_id number;
EXEC :department_id := 50;

-- Employees who earn more than their managers
SELECT
  e1.employee_id,
  e1.salary,
  e2.employee_id,
  e2.salary
```

```
FROM
  employees e1,
  employees e2
WHERE
  e1.department_id = :department_id
  AND e1.manager_id = e2.employee_id
  AND e1.salary > e2.salary;

no rows selected


Execution Plan
----------------------------------------------------------
Plan hash value: 2254211361

-------------------------------------------------------------------------------------------------
| Id  | Operation                    | Name             | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                  |     1 |    23 |     6  (17)| 00:00:01 |
|*  1 |  HASH JOIN                   |                  |     1 |    23 |     6  (17)| 00:00:01 |
|   2 |   TABLE ACCESS BY INDEX ROWID| EMPLOYEES        |    10 |   150 |     2   (0)| 00:00:01 |
|*  3 |    INDEX RANGE SCAN          | EMP_DEPARTMENT_IX |    10 |       |     1   (0)| 00:00:01 |
|   4 |   TABLE ACCESS FULL          | EMPLOYEES        |   107 |   856 |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------

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

   1 - access("E1"."MANAGER_ID"="E2"."EMPLOYEE_ID")
       filter("E1"."SALARY">"E2"."SALARY")
   3 - access("E1"."DEPARTMENT_ID"=TO_NUMBER(:DEPARTMENT_ID))


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

TKPROF Only Tells Part Of The Story

A better way to review a query plan is to trace a session and use the `tkprof` utility to format the trace file. Complete instructions for using the tkprof utility can be found in the Performance Tuning Guide. Unfortunately, tracing a session requires privileges that an application developer may not have. Also, trace files are generated in Oracle-owned directories on the database server; application developers may not have access to these directories. In the following example, I use `ALTER SESSION` in a SQL*Plus session. However, a SQL*Plus session may not use the same query plan that is used when the query is executed from within your application. Nevertheless, the example below proves that `EXPLAIN PLAN` and autotrace were

lying; the real query plan is very different from those produced by `EXPLAIN PLAN` and `autotrace` in the previous sections.

```
ALTER SESSION SET statistics_level=ALL;
ALTER SESSION SET tracefile_identifier=TEST;
ALTER SESSION SET sql_trace=TRUE;
```

While `tkprof` portrays a true story, it only tells the partial story. What we really need is the estimates side by side with the actual numbers.

```
call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        1      0.00       0.00          0         14          0           0
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        3      0.00       0.00          0         14          0           0

Misses in library cache during parse: 0
Optimizer mode: ALL_ROWS
Parsing user id: 180

Rows     Row Source Operation
-------  ---------------------------------------------------
      0  HASH JOIN  (cr=14 pr=0 pw=0 time=1681 us)
     45   TABLE ACCESS FULL EMPLOYEES (cr=7 pr=0 pw=0 time=446 us)
    107   TABLE ACCESS FULL EMPLOYEES (cr=7 pr=0 pw=0 time=25 us)
```

Reviewing The Complete Picture With DBMS_XPLAN

The best way to review a query plan is using DBMS_XPLAN.DISPLAY_CURSOR which displays optimizer estimates side by side with actual execution metrics. This requires access to data dictionary tables: V$SESSION, V$SQL, and V$SQL_PLAN_STATISTICS_ALL. In order for Oracle to collect and display row counts and execution metrics for each step in a query plan, it is critical that STATISTICS_LEVEL be set to `ALL`; this can be done at the session level or at the system level. In the example below, we have broken the output into separate tables since it does not fit within the margins.

```
ALTER SESSION SET statistics_level=ALL;

VARIABLE department_id number;
EXEC :department_id := 50;

SELECT
  e1.employee_id,
  e1.salary,
  e2.employee_id,
  e2.salary
FROM
  employees e1,
  employees e2
WHERE
  e1.department_id = :department_id
  AND e1.manager_id = e2.employee_id
  AND e1.salary > e2.salary;

SELECT *
FROM TABLE (DBMS_XPLAN.display_cursor (
  NULL, -- this automatically uses the sql_id of the last SQL*Plus query
  NULL, -- this automatically uses the child_number of the last SQL*Plus query
```

```
  'TYPICAL IOSTATS LAST +PEEKED_BINDS'));

SQL_ID  g9n0ar49yr8hc, child number 1
-------------------------------------

Plan hash value: 468575080

-----------------------------------------------------------------------------------------
| Id  | Operation          | Name      | Starts | E-Rows |E-Bytes| Cost (%CPU)| E-Time   |
-----------------------------------------------------------------------------------------
|*  1 |  HASH JOIN         |           |      1 |      2 |    46 |     7  (15)| 00:00:01 |
|*  2 |   TABLE ACCESS FULL| EMPLOYEES |      1 |     45 |   675 |     3   (0)| 00:00:01 |
|   3 |   TABLE ACCESS FULL| EMPLOYEES |      1 |    107 |   856 |     3   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------
---------------------------------------------------------------------
| Id  | Operation          | Name      | A-Rows |   A-Time   | Buffers |
---------------------------------------------------------------------
|*  1 |  HASH JOIN         |           |      0 |00:00:00.01 |      14 |
|*  2 |   TABLE ACCESS FULL| EMPLOYEES |     45 |00:00:00.01 |       7 |
|   3 |   TABLE ACCESS FULL| EMPLOYEES |    107 |00:00:00.01 |       7 |
---------------------------------------------------------------------

Peeked Binds (identified by position):
--------------------------------------

   1 - (NUMBER): 50

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

   1 - access("E1"."MANAGER_ID"="E2"."EMPLOYEE_ID")
       filter("E1"."SALARY">"E2"."SALARY")
   2 - filter("E1"."DEPARTMENT_ID"=:DEPARTMENT_ID)
```

Interpreting Query Execution plans

Finally we come to the business of interpreting query execution plans: what does it all mean? There is a tremendous amount of information available in the Oracle documentation. Chapter 11 *The Query Optimizer* in the *Performance Tuning Guide* explains *access paths* and *join methods* while Chapter 12 *Using Explain Plan* provides many examples of query plans. In the rest of this paper, I will focus on two critical areas that are not adequately covered in the documentation: determining the order of execution of each step and determining which steps consume the most resources. Here is an example of a fairly complex query plan produced by `DBMS_XPLAN.DISPLAY_CURSOR`; only a few columns are shown in the interests of clarity.

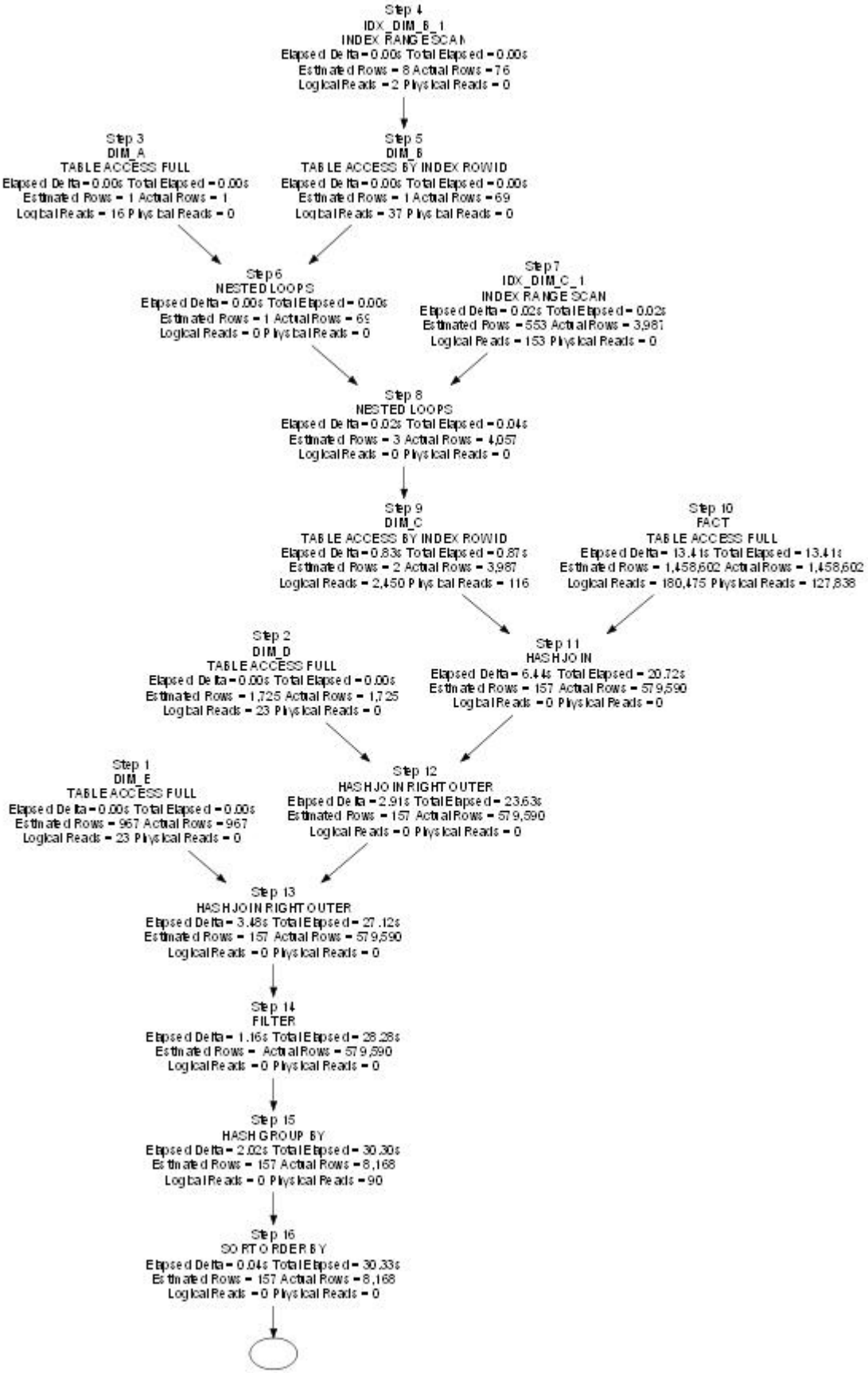
```
-------------------------------------------------------------------------
|  Id | Operation                           | Name        | A-Time     |
-------------------------------------------------------------------------
|   1 |  SORT ORDER BY                      |             |00:00:30.33 |
|   2 |   HASH GROUP BY                     |             |00:00:30.30 |
|*  3 |    FILTER                           |             |00:00:28.28 |
|*  4 |     HASH JOIN RIGHT OUTER           |             |00:00:27.12 |
|   5 |      TABLE ACCESS FULL              | DIM_E       |00:00:00.01 |
|*  6 |      HASH JOIN RIGHT OUTER          |             |00:00:23.63 |
|   7 |       TABLE ACCESS FULL             | DIM_D       |00:00:00.01 |
|*  8 |       HASH JOIN                     |             |00:00:20.72 |
|   9 |        TABLE ACCESS BY INDEX ROWID  | DIM_C       |00:00:00.87 |
|  10 |         NESTED LOOPS                |             |00:00:00.04 |
|  11 |          NESTED LOOPS               |             |00:00:00.01 |
|* 12 |           TABLE ACCESS FULL         | DIM_A       |00:00:00.01 |
|* 13 |           TABLE ACCESS BY INDEX ROWID| DIM_B      |00:00:00.01 |
|* 14 |            INDEX RANGE SCAN         | IDX_DIM_B_1 |00:00:00.01 |
|* 15 |          INDEX RANGE SCAN           | IDX_DIM_C_1 |00:00:00.02 |
|* 16 |        TABLE ACCESS FULL            | FACT        |00:00:13.41 |
-------------------------------------------------------------------------
```

The critical point is that the Id column does not indicate the order in which operations are executed, instead it is the indentation of operations that indicates the right order. The following rule is contained in the Oracle documentation: *"The execution order in EXPLAIN PLAN output begins with the line that is the furthest indented to the right. The next step is the parent of that line. If two lines are indented equally, then the top line is normally executed first."* but this is not precise enough. The following statement is more precise, if a lot harder to read: *"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."*. If you concentrate really hard, you'll eventually figure out that the operations are performed in the following sequence: 5, 7, 12, 14, 13, 11, 15, 10, 9, 16, 8, 6, 4, 3, 2, and 1.

I have to admit that I find tabular query execution plans—such as that shown above—not very easy to read, especially when many tables are involved. The last time I was trying to make sense of such a plan, a colleague suggested that I use a sheet of paper as a makeshift ruler. Another problem is that the elapsed execution times that are listed in the plans are cumulative; this makes it difficult to identify the time-consuming steps.

A graphical query plan such as the one shown on the next page is much easier to read. In addition, it shows execution times for individual operations, not cumulative execution times. The PL/SQL code that produced it is shown in the following pages. It produces commands—in the "dot" language—for a graphing tool called Graphviz that can be downloaded from http://www.graphviz.org/. Here is an example; it shows abbreviated versions of the commands needed to produce the graph on the next page.

```
digraph a {
"5" [label="Step 1\nDIM_E",shape=plaintext]
"7" [label="Step 2\nDIM_D",shape=plaintext]
"12" [label="Step 3\nDIM_A",shape=plaintext]
"14" [label="Step 4\nIDX_DIM_B_1",shape=plaintext]
"13" [label="Step 5\nDIM_B",shape=plaintext]
"11" [label="Step 6\nNESTED LOOPS",shape=plaintext]
"15" [label="Step 7\nIDX_DIM_C_1",shape=plaintext]
"10" [label="Step 8\nNESTED LOOPS",shape=plaintext]
"9" [label="Step 9\nDIM_C",shape=plaintext]
"16" [label="Step 10\nFACT",shape=plaintext]
"8" [label="Step 11\nHASH JOIN",shape=plaintext]
"6" [label="Step 12\nHASH JOIN RIGHT OUTER",shape=plaintext]
"4" [label="Step 13\nHASH JOIN RIGHT OUTER",shape=plaintext]
"3" [label="Step 14\nFILTER",shape=plaintext]
"2" [label="Step 15\nHASH GROUP BY",shape=plaintext]
"1" [label="Step 16\nSORT ORDER BY",shape=plaintext]
"1"->"";
"2"->"1";
"3"->"2";
"4"->"3";
"5"->"4";
"6"->"4";
"7"->"6";
"8"->"6";
"9"->"8";
"10"->"9";
"11"->"10";
"12"->"11";
"13"->"11";
"14"->"13";
"15"->"10";
"16"->"8";
};
```


Step 4
IDX_DIM_B_1
INDEX RANGE SCAN
Elapsed Delta = 0.00s Total Elapsed = 0.00s
Estimated Rows = 8 Actual Rows = 76
Logical Reads = 2 Physical Reads = 0
Step 3
DIM_A
TABLE ACCESS FULL
Elapsed Delta = 0.00s Total Elapsed = 0.00s
Estimated Rows = 1 Actual Rows = 1
Logical Reads = 16 Physical Reads = 0
Step 5
DIM_B
TABLE ACCESS BY INDEX ROWID
Elapsed Delta = 0.00s Total Elapsed = 0.00s
Estimated Rows = 1 Actual Rows = 69
Logical Reads = 37 Physical Reads = 0
Step 6
NESTED LOOPS
Elapsed Delta = 0.00s Total Elapsed = 0.00s
Estimated Rows = 1 Actual Rows = 69
Logical Reads = 0 Physical Reads = 0
Step 7
IDX_DIM_C_1
INDEX RANGE SCAN
Elapsed Delta = 0.02s Total Elapsed = 0.02s
Estimated Rows = 553 Actual Rows = 3,987
Logical Reads = 153 Physical Reads = 0
Step 8
NESTED LOOPS
Elapsed Delta = 0.02s Total Elapsed = 0.04s
Estimated Rows = 3 Actual Rows = 4,057
Logical Reads = 0 Physical Reads = 0
Step 9
DIM_C
TABLE ACCESS BY INDEX ROWID
Elapsed Delta = 0.83s Total Elapsed = 0.87s
Estimated Rows = 2 Actual Rows = 3,987
Logical Reads = 2,450 Physical Reads = 116
Step 10
FACT
TABLE ACCESS FULL
Elapsed Delta = 13.41s Total Elapsed = 13.41s
Estimated Rows = 1,458,602 Actual Rows = 1,458,602
Logical Reads = 180,475 Physical Reads = 127,838
Step 2
DIM_D
TABLE ACCESS FULL
Elapsed Delta = 0.00s Total Elapsed = 0.00s
Estimated Rows = 1,725 Actual Rows = 1,725
Logical Reads = 23 Physical Reads = 0
Step 11
HASH JOIN
Elapsed Delta = 6.44s Total Elapsed = 20.72s
Estimated Rows = 157 Actual Rows = 579,590
Logical Reads = 0 Physical Reads = 0
Step 1
DIM_E
TABLE ACCESS FULL
Elapsed Delta = 0.00s Total Elapsed = 0.00s
Estimated Rows = 967 Actual Rows = 967
Logical Reads = 23 Physical Reads = 0
Step 12
HASH JOIN RIGHT OUTER
Elapsed Delta = 2.91s Total Elapsed = 23.63s
Estimated Rows = 157 Actual Rows = 579,590
Logical Reads = 0 Physical Reads = 0
Step 13
HASH JOIN RIGHT OUTER
Elapsed Delta = 3.48s Total Elapsed = 27.12s
Estimated Rows = 157 Actual Rows = 579,590
Logical Reads = 0 Physical Reads = 0
Step 14
FILTER
Elapsed Delta = 1.16s Total Elapsed = 28.28s
Estimated Rows = Actual Rows = 579,590
Logical Reads = 0 Physical Reads = 0
Step 15
HASH GROUP BY
Elapsed Delta = 2.02s Total Elapsed = 30.30s
Estimated Rows = 157 Actual Rows = 8,168
Logical Reads = 0 Physical Reads = 90
Step 16
SORT ORDER BY
Elapsed Delta = 0.04s Total Elapsed = 30.33s
Estimated Rows = 157 Actual Rows = 8,168
Logical Reads = 0 Physical Reads = 0

The source of the information shown is V$SQL_PLAN_STATISTICS_ALL which is the same as that used by DBMS_XPLAN. A PL/SQL function is called recursively in order to produce the information that is needed. Assuming that you have installed Graphviz on your computer, you can use the following command to produce a graphical query plan from the output (spool.dot) of the code. Various output formats are available; the example shown below uses the JPG format.

```
dot -Tjpg -oplan.jpg plan.dot
```

Here is the complete source code for the enhanced_plan package.

```
Copyright 2009 Iggy Fernandez

This program is free software: you can redistribute it and/or modify it under the
terms of the GNU General Public License as published by the Free Software Foundation,
either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY
WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.  See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this
program.  If not, see <http://www.gnu.org/licenses/>.

CREATE OR REPLACE TYPE enhanced_plan_type
AS OBJECT
(
  execution_id NUMBER,
  operation VARCHAR2 (120),
  options VARCHAR2 (120),
  object_owner VARCHAR2 (30),
  object_name VARCHAR2 (30),
  id NUMBER,
  parent_id NUMBER,
  cardinality NUMBER,
  last_output_rows NUMBER,
  last_logical_reads NUMBER,
  last_disk_reads NUMBER,
  last_elapsed_time NUMBER,
  delta_elapsed_time NUMBER
);
/

CREATE OR REPLACE TYPE enhanced_plan_table
AS TABLE OF enhanced_plan_type
/

CREATE OR REPLACE PACKAGE enhanced_plan
AS
  FUNCTION plan
  (
    sql_id_in VARCHAR2,
    child_number_in NUMBER,
    parent_id_in NUMBER DEFAULT 0
  )
  RETURN enhanced_plan_table PIPELINED;
END enhanced_plan;
/
```

```
CREATE OR REPLACE PACKAGE BODY enhanced_plan AS

  FUNCTION PLAN
  (
    sql_id_in VARCHAR2,
    child_number_in NUMBER,
    parent_id_in NUMBER DEFAULT 0
  )
  RETURN enhanced_plan_table PIPELINED
  IS

    parent_row enhanced_plan_type := enhanced_plan_type
    (
      NULL, NULL, NULL, NULL, NULL, NULL, NULL,
      NULL, NULL, NULL, NULL, NULL, NULL
    );

    child_row enhanced_plan_type := enhanced_plan_type
    (
      NULL, NULL, NULL, NULL, NULL, NULL, NULL,
      NULL, NULL, NULL, NULL, NULL, NULL
    );

    execution_id NUMBER := 1;

    CURSOR parent_cursor IS
    WITH
    parent_statistics AS
    (
      SELECT
        operation,
        options,
        object_owner,
        object_name,
        id,
        parent_id,
        cardinality,
        last_output_rows,
        last_cr_buffer_gets + last_cu_buffer_gets
          AS last_logical_reads,
        last_disk_reads,
        last_elapsed_time / 1000000
          AS last_elapsed_time
      FROM
        v$sql_plan_statistics_all
       WHERE
        sql_id = sql_id_in
        AND child_number = child_number_in
        AND parent_id = parent_id_in
    ),
    child_statistics AS
    (
      SELECT
```

```
      parent_id,
      SUM (last_cr_buffer_gets + last_cu_buffer_gets)
        AS last_logical_reads,
      SUM (last_disk_reads) AS last_disk_reads,
      SUM (last_elapsed_time) / 1000000
        AS last_elapsed_time
    FROM
      v$sql_plan_statistics_all
    WHERE sql_id = sql_id_in
    AND child_number = child_number_in
    GROUP BY parent_id
  )
  SELECT
    p.operation,
    p.options,
    p.object_owner,
    p.object_name,
    p.ID,
    p.parent_id,
    p.cardinality,
    p.last_output_rows,
    p.last_logical_reads - NVL (c.last_logical_reads, 0)
      AS last_logical_reads,
    p.last_disk_reads - NVL (c.last_disk_reads, 0)
      AS last_disk_reads,
    p.last_elapsed_time AS last_elapsed_time,
    (p.last_elapsed_time - NVL (c.last_elapsed_time, 0))
      AS delta_elapsed_time
  FROM parent_statistics p, child_statistics c
  WHERE p.ID = c.parent_id(+)
  ORDER BY p.ID;

  CURSOR child_cursor IS
  SELECT
    operation,
    options,
    object_owner,
    object_name,
    ID,
    parent_id,
    cardinality,
    last_output_rows,
    last_logical_reads,
    last_disk_reads,
    last_elapsed_time,
    delta_elapsed_time
  FROM TABLE (enhanced_plan.plan (
    sql_id_in,
    child_number_in,
    parent_row.ID
  ));

BEGIN
```

```
    OPEN parent_cursor;
    LOOP
      FETCH parent_cursor
      INTO
        parent_row.operation,
        parent_row.options,
        parent_row.object_owner,
        parent_row.object_name,
        parent_row.ID,
        parent_row.parent_id,
        parent_row.cardinality,
        parent_row.last_output_rows,
        parent_row.last_logical_reads,
        parent_row.last_disk_reads,
        parent_row.last_elapsed_time,
        parent_row.delta_elapsed_time;
      EXIT WHEN parent_cursor%NOTFOUND;
      OPEN child_cursor;
      LOOP
        FETCH child_cursor
        INTO
          child_row.operation,
          child_row.options,
          child_row.object_owner,
          child_row.object_name,
          child_row.ID,
          child_row.parent_id,
          child_row.cardinality,
          child_row.last_output_rows,
          child_row.last_logical_reads,
          child_row.last_disk_reads,
          child_row.last_elapsed_time,
          child_row.delta_elapsed_time;
        EXIT WHEN child_cursor%NOTFOUND;
        child_row.execution_id := execution_id;
        execution_id := execution_id + 1;
        PIPE ROW (child_row);
      END LOOP;
      CLOSE child_cursor;
      parent_row.execution_id := execution_id;
      execution_id := execution_id + 1;
      PIPE ROW (parent_row);
    END LOOP;
    CLOSE parent_cursor;

  END plan;

END enhanced_plan;
/

SET linesize 1000
SET trimspool on
SET pagesize 0
SET echo off
```

```
SET heading off
SET feedback off
SET verify off
SET time off
SET timing off
SET sqlblanklines on

DEFINE sql_id = &sql_id
DEFINE child_number = &child_number

SPOOL plan.dot

WITH

plan_table AS
(
  SELECT
    *
  FROM
    TABLE (enhanced_plan.plan (
      '&sql_id',
      &child_number
    ))
)

SELECT
  'digraph a {'
FROM
  DUAL

UNION ALL

SELECT
  '"'
  || id
  || '" [label="Step '
  || execution_id
  || '\n'
  || CASE WHEN object_name IS NULL
    THEN ('')
    ELSE (object_name || '\n')
    END
  || CASE WHEN options IS NULL
    THEN (operation || '\n')
    ELSE (operation || ' ' || options || '\n')
    END
  || 'Elapsed Delta = '
  || TRIM (TO_CHAR (delta_elapsed_time, '999,999,990.00'))
  || 's'
  || ' Total Elapsed = '
  || TRIM (TO_CHAR (last_elapsed_time, '999,999,990.00'))
  || 's\n'
  || 'Estimated Rows = '
  || TRIM (TO_CHAR (cardinality, '999,999,999,999,990'))
```

```
  || ' Actual Rows = '
  || TRIM (TO_CHAR (last_output_rows, '999,999,999,999,990'))
  || '\n'
  || 'Logical Reads = '
  || TRIM (TO_CHAR (last_logical_reads, '999,999,999,999,990'))
  || ' Physical Reads = '
  || TRIM (TO_CHAR (last_disk_reads, '999,999,999,999,990'))
  || '",shape=plaintext]' op
FROM
  plan_table

UNION ALL

SELECT
  edge
FROM
  (
    SELECT
      parent_id,
      '"' || id || '"' || '->' || '"' || PRIOR id || '"' || ';'
        AS edge
    FROM
      plan_table
      START WITH parent_id = 0
      CONNECT BY parent_id = PRIOR id
  )
  WHERE
    parent_id IS NOT NULL

UNION ALL

SELECT
  '};'
FROM
  DUAL;

SPOOL off
```

Concluding Remarks

In this paper, I have demonstrated that the output of `DBMS_XPLAN.DISPLAY_CURSOR('&sql_id', '&child_number', 'TYPICAL IOSTATS LAST +PEEKED BINDS')` with `STATISTICS_LEVEL` set to `ALL` is the most appropriate place to start when reviewing query performance. I have also demonstrated an alternative graphical method of reviewing query plans. Please send any comments or questions to iggy_fernandez@hotmail.com.

About Me

I am an Oracle DBA with Database Specialists and have fifteen years of experience in Oracle database administration. I am the editor of the quarterly Journal of the Northern California Oracle Users Group (NoCOUG) and the author of *Beginning Oracle Database 11g Administration* (Apress, 2009). You can reach me at iggy_fernandez@hotmail.com or visit my blog.