Interpreting Explain Plan (10g and Above) (Doc ID 1616894.1)

To Bottom

Modified: 27-Feb-2014 Type: BULLETIN

In this Document

APPLIES TO:

Oracle Database - Enterprise Edition - Version 10.1.0.2 and later
Oracle Communications MetaSolv Solution - Version 6.2.1 to 6.2.1 [Release 6.2.0]
Information in this document applies to any platform.

PURPOSE

This article outlines how to interpret explain plans. It is based upon an earlier document which was written in the 8i timeframe and updated to reflect more recent developments. For legacy information see:

> Document 46234.1 Interpreting Explain Plan

The manual also provides useful material:

> Oracle® Database SQL Tuning Guide
> 12c Release 1 (12.1)
> E15858-15
> Chapter 8 Generating and Displaying Execution Plans

http://docs.oracle.com/cd/E16655_01/server.121/e15858/tgsql_genplan.htm#TGSQL271

Chapter 9 Reading Execution Plans
http://docs.oracle.com/cd/E16655_01/server.121/e15858/tgsql_interp.htm#TGSQL94618

SCOPE

The audience for this article is anyone interested in undertanding the basics of reading explain plan

DETAILS

What is an explain plan?

An explain plan is a representation of the access path that is taken when a query is executed within Oracle. Query processing can be divided into 7 phases:

1. Syntactic - checks the syntax of the query
2. Semantic- checks that all objects exist and are accessible
3. View Merging - rewrites query as join on base tables as opposed to using views
4. Statement Transformation - rewrites query transforming some complex constructs into simpler ones where appropriate (e.g. subquery unnesting, in/or transformation). Some transformations use rules while others are costed based upon statistics.
5. Optimization - determines the optimal access path for the query to take. The Cost Based Optimizer (CBO) uses statistics to analyze the relative costs of accessing objects.
6. Query Evaluation Plan(QEP) Generation
7. QEP Execution

Steps [1]-[6] are sometimes grouped under the term 'Parsing'
Step [7] is the execution of the statement.

The explain plan is a representation of the access path produced in step 6.

Once the access path has been decided upon, it is stored in the library cache together with the statement itself. Queries are stored in the library cache based upon a hashed representation of the query. When looking for a statement in the library cache, we first apply a hashing algorithm to the current statement and then look for this hash value in the library cache. This access path will be used until the query is re-parsed.

Terminology

Row Source - A row source is a software function that implements specific operations (such as a table scan or a hash join) and returns a set of rows.
Predicates - The where clause of a query
Tuples - rows
Driving Table - This is the row source that we use to seed the query. If this returns a lot of rows then this can have a negative effect on all subsequent operations
Probed Table - This is the object we look-up data in after we have retrieved relevant key data from the driving table.

How does Oracle access data?

At the physical level Oracle reads blocks of data.The smallest amount of data read is a single Oracle block, the largest is constrained by operating system limits (and multi-block i/o). Logically Oracle finds the data to read by using the following methods:

Full Table Scan (FTS)
Index Look-up (unique & non-unique)
Index Full Scan
Index Fast Full Scan
Rowid

Examples

For convenience the example explain plan output in this article was created using the autotrace feature of SQL*Plus. Statistics_level was set to typical. There are many different ways in which you can display an explain plan, a number of which are outlined in the following article:

Document 235530.1 How to Obtain a Formatted Explain Plan - Recommended Methods

Although different methods of displaying plans may look slightly different, the basic information the plan is produced from is the same.

Explain plan Sections

When you generate an explain plan a number of sections will be displayed dependent on the query that the plan has been generated from and the features that the execution plan uses. All execution plans will have a main plan table showing the operations taken to execute the query and possibly including a variety of other information such as "Plan hash value", predicted row counts, cost estimates for each step etc. for example:

```
Execution Plan
----------------------------------------------------------
Plan hash value: 2872589290

--------------------------------------------------------------------------
| Id  | Operation         | Name | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |      |    14 |   532 |     3   (0)| 00:00:01 |
|   1 |  TABLE ACCESS FULL| EMP  |    14 |   532 |     3   (0)| 00:00:01 |
--------------------------------------------------------------------------
```

If the query has a where clause, the predicates will be output :

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

   2 - filter("DEPT"."DNAME"='ACCOUNTING' OR "DEPT"."DNAME"='OPERATIONS' OR
              "DEPT"."DNAME"='RESEARCH' OR "DEPT"."DNAME"='SALES')
   4 - access("EMP"."DEPTNO"="DEPT"."DEPTNO")
       filter("EMP"."DEPTNO"="DEPT"."DEPTNO")
```

Predicates are either access predicates (which have been used to access data in underlying objects) or filter predicates (which are used to restrict the rows returned from a particular step and refine the results). Note that transformations may display predicates in a slightly different form to those presented in the original query.

On occasion, a "Note" section will be output which includes useful information regarding the plan. The section may indicate (among others) whether the database used a SQL Plan Baseline, outline or SQL profile for the query, if dynamic sampling has been used (with the level), if the optimizer used statistics feedback to adjust its cost estimates for the second execution of the query, if the plan is an adaptive plan or if the query is wholly remote. For example:

```
Note
-----
   - SQL plan baseline SYS_SQL_PLAN_fcc170b0a62d0f4d used for this statement

Note
-----
   - dynamic statistics used: dynamic sampling (level=2)

Note
-----
   -  statistics feedback used for this statement

Note
-----
   - this is an adaptive plan

Note
-----
   - fully remote statement
```

If a query is partly remote then a section may be included that shows the query that has been sent to the remote site:

```
Remote SQL Information (identified by operation id):
----------------------------------------------------

   4 - SELECT "DEPTNO","DNAME" FROM "DEPT" "A" (accessing 'LOOP_LINK' )
```

Explain plan Hierarchy

If you consider the following simple explain plan:

```
SQL> set autot traceonly explain
SQL> select * from emp;

Execution Plan
----------------------------------------------------------
Plan hash value: 2872589290

--------------------------------------------------------------------------
| Id  | Operation         | Name | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |      |    14 |   532 |     3   (0)| 00:00:01 |
|   1 |  TABLE ACCESS FULL| EMP  |    14 |   532 |     3   (0)| 00:00:01 |
--------------------------------------------------------------------------
```

The steps in the explain plan are indented to indicate the hierarchy of operations and which steps are dependent on which other steps. When looking at an indented plan, to find which operation is executed first, examine the Operation column. In this column, the rightmost (ie most indented) uppermost operation is the first thing that is executed. In other words, look down the Operation column starting from the top until you find the operation that is indented the most. This is executed first. In this example this would be the operation with Id=1. A more detailed explanation of this much simplified description can be found below.

In the example, TABLE ACCESS FULL EMP is the first operation that will occur. This statement means we are doing a full table scan of table EMP. When this operation completes then the resultant row source is passed up to the next level of the query for processing. In this case it is the SELECT STATEMENT which is the top of the query.

The other columns in the explain plan provide various pieces of useful information in determining why the plan was chosen:

- Rows - this tells us the estimated number of rows that the optimizer expects this line of the execution plan to return
- Bytes - this tells us the estimated number of bytes that the optimizer expects this line of the execution plan will return
- Cost (%CPU) - this is the optimizer's estimation of the 'cost' and %CPU of the query. The cost allows the optimizer to compare the estimated performance of different plans with each other.
- Time - this is the optimizer's estimation of the duration of each step of the query

More about plans and the order of execution

To understand plans and the order of execution, it is necessary to understand the PARENT -- CHILD relationships involved:

```
PARENT
  FIRST  CHILD
  SECOND CHILD
```

In this example, the FIRST CHILD is executed first followed by the SECOND CHILD, then the PARENT collates the output in some way.

A more complex case is:

```
PARENT1
```

```
    FIRST  CHILD
      FIRST  GRANDCHILD
    SECOND CHILD
```

Here the same principles apply, the FIRST GRANDCHILD is the initial operation then the FIRST CHILD followed by the SECOND CHILD and finally the PARENT collates the output. These principles can be applied to real operations as in the examples below.
Consider the following query:

Example 1

```
set autotrace traceonly explain

select ename,dname
  from emp, dept
 where emp.deptno=dept.deptno
   and dept.dname in ('ACCOUNTING','RESEARCH','SALES','OPERATIONS');
Execution Plan
----------------------------------------------------------
Plan hash value: 2865896559

---------------------------------------------------------------------------------------
| Id  | Operation                    | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
---------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |         |    14 |   308 |     5   (0)| 00:00:01 |
|   1 |  MERGE JOIN                  |         |    14 |   308 |     5   (0)| 00:00:01 |
|*  2 |   TABLE ACCESS BY INDEX ROWID| DEPT    |     4 |    52 |     2   (0)| 00:00:01 |
|   3 |    INDEX FULL SCAN           | PK_DEPT |     4 |       |     1   (0)| 00:00:01 |
|*  4 |   SORT JOIN                  |         |    14 |   126 |     3   (0)| 00:00:01 |
|   5 |    TABLE ACCESS FULL         | EMP     |    14 |   126 |     3   (0)| 00:00:01 |
---------------------------------------------------------------------------------------

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

   2 - filter("DEPT"."DNAME"='ACCOUNTING' OR "DEPT"."DNAME"='OPERATIONS' OR
              "DEPT"."DNAME"='RESEARCH' OR "DEPT"."DNAME"='SALES')
   4 - access("EMP"."DEPTNO"="DEPT"."DEPTNO")
       filter("EMP"."DEPTNO"="DEPT"."DEPTNO")
```

What follows is a walk-through of the plan above:

Execution starts with: ID=3 and it gets there as follows:
Starting with ID=0:

```
------------------------------------------------
| Id  | Operation                    | Name    |
------------------------------------------------
|   0 | SELECT STATEMENT             |         |
|   1 |  MERGE JOIN                  |         |
|*  2 |   TABLE ACCESS BY INDEX ROWID| DEPT    |
|   3 |    INDEX FULL SCAN           | PK_DEPT |
|*  4 |   SORT JOIN                  |         |
|   5 |    TABLE ACCESS FULL         | EMP     |
------------------------------------------------
```

ID=0 has no operation above it, so it has no parent but it has 1 child.
ID=0 is the parent of ID=1 and is dependent upon it for rows.You can tell it is the parent because the child is indented.
So ID=1 must be executed prior to ID=0

Moving on to ID=1:

```
|   0 | SELECT STATEMENT             |         |
|   1 |  MERGE JOIN                  |         |
|*  2 |   TABLE ACCESS BY INDEX ROWID| DEPT    |

|*  4 |   SORT JOIN                  |         |
```

As before, ID=1 is the child of ID=0.
From the indentation, ID=2 and ID=4 are indented at the same level beneath ID=1. Thus ID=1 is the parent of ID=2 and ID=4 and is dependent upon them for rows. So ID=2 and ID=4 must be executed prior to ID=1

Moving on to ID=2:

```
|   1 |  MERGE JOIN                  |         |
|*  2 |   TABLE ACCESS BY INDEX ROWID| DEPT    |
|   3 |    INDEX FULL SCAN           | PK_DEPT |
```

ID=2 is the first child of ID=1.
From the indentation, ID=2 is the parent of ID=3 and is dependent upon it for rows. So ID=3 must be executed prior to ID=2. Moving on to ID=3:

```
|   1 |  MERGE JOIN                  |         |
|*  2 |   TABLE ACCESS BY INDEX ROWID| DEPT    |
|   3 |    INDEX FULL SCAN           | PK_DEPT |
```

ID=3 is the (only) child of ID=2.
ID=3 has no child operations. This means that ID=3 is the first step that is executed by the query. Rows are provided to ID=2 from this step.

ID=1 and ID=0 are also dependent on ID=3. Once ID=3 has produced rows, they are passed to ID=2 and that step uses them for whatever operation it is performing. It then provides the processed rows to its parent and so on up the tree. This means that ID=2 is the second step that is executed. ID=1 is not executed next because it has 2 inputs. It needs both of these to be accessed before it can start to operate so now lets look at the second child of ID=1, ID=4:

```
|   1 |  MERGE JOIN                  |         |


|*  4 |   SORT JOIN                  |         |
|   5 |    TABLE ACCESS FULL         | EMP     |
```

ID=4 is the second child of ID=1.
ID=4 is the parent of ID=5 and is dependent upon it for rows. ID=5 must be executed prior to ID=4. This means that ID=5 is the third step that is executed followed by ID=4.

```
|   1 |  MERGE JOIN                  |         |


|*  4 |   SORT JOIN                  |         |
|   5 |    TABLE ACCESS FULL         | EMP     |
```

Once ID=1 has inputs from both of it's children it can execute. ID=1 processes the rows it receives from its dependent steps (ID=2 & ID=4) and returns them to its parent ID=0.

ID=0 returns the rows to the user.

A shortened summary of this is:

To find the execution order:

- Start with ID=0: SELECT STATEMENT but this is dependent on it's child objects
- So it looks at it's first child step: ID=1 MERGE JOIN but this is dependent on it's child objects
- So it looks at it's first child step: ID=2   TABLE ACCESS BY INDEX ROWID DEPT but this is dependent on it's child object
- So it looks at it's only child step: ID=3   INDEX FULL SCAN PK_DEPT. This has no children so this is executed.
- Rows from ID=3 are fed back to ID=2
- Rows from ID=2 are fed back to ID=1 but this has 2 children so the other child ID=4 needs to be explored
- So it looks at it's second child step: ID=4   SORT JOIN but this is dependent on it's child object
- So it looks at it's only child step: ID=5   TABLE ACCESS FULL EMP This has no children so this is executed.
- Rows from ID=5 are fed back to ID=4
- Rows from ID=4 are fed back to ID=1. Since both children have now suplied rows, ID=1 can be executed.
- Rows from ID=1 are fed back to ID=0
- Rows from ID=0 are fed back to the client
- Rows are returned to the parent step(s) until finished

Execution order is 3,2,4,5,1,0

Example 2

```
select /*+ ordered USE_NL(dept) */ ename,dname
  from emp, dept
 where emp.deptno=dept.deptno
   and dept.dname in ('ACCOUNTING','RESEARCH','SALES','OPERATIONS');

Execution Plan
----------------------------------------------------------
Plan hash value: 196120631

----------------------------------------------------------------------------------------
| Id  | Operation                    | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |         |    14 |   308 |    17   (0)| 00:00:01 |
|   1 |  NESTED LOOPS                |         |       |       |            |          |
|   2 |   NESTED LOOPS               |         |    14 |   308 |    17   (0)| 00:00:01 |
|   3 |    TABLE ACCESS FULL         | EMP     |    14 |   126 |     3   (0)| 00:00:01 |
|*  4 |    INDEX UNIQUE SCAN         | PK_DEPT |     1 |       |     0   (0)| 00:00:01 |
|*  5 |   TABLE ACCESS BY INDEX ROWID| DEPT    |     1 |    13 |     1   (0)| 00:00:01 |
----------------------------------------------------------------------------------------

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

   4 - access("EMP"."DEPTNO"="DEPT"."DEPTNO")
   5 - filter("DEPT"."DNAME"='ACCOUNTING' OR "DEPT"."DNAME"='OPERATIONS' OR
              "DEPT"."DNAME"='RESEARCH' OR "DEPT"."DNAME"='SALES')
```

In this example execution starts with ID=3 and we can get there as follows:

To find the execution order:

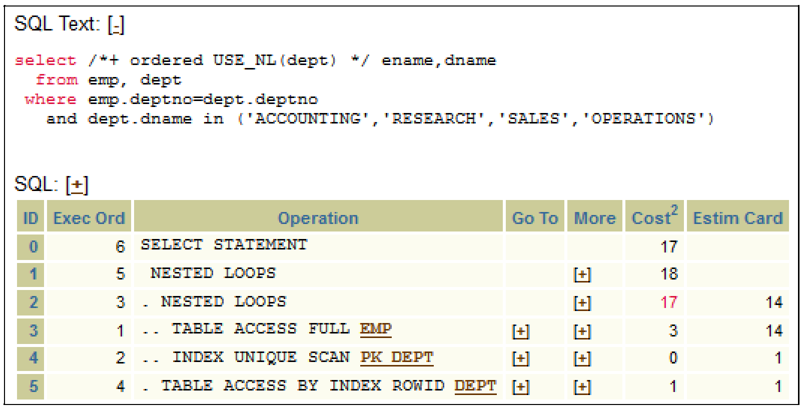
- Start with ID=0 SELECT STATEMENT but this is dependent on it's child objects
- So it looks at it's first child step: ID=1 NESTED LOOPS but this is dependent on it's child objects
- So it executes it's first child step: ID=2 NESTED LOOPS but this is dependent on it's child objects
- So it executes it's first child step: ID=3 TABLE ACCESS (FULL) OF 'EMP' This has no child object so this is executed.
- Rows from ID=3 are fed back to ID=2
- Because ID=2 is a NESTED LOOPS join, the first input is used to drive the join
- ID=2 uses the rows to drive the lookup of it's second child ID=4: INDEX UNIQUE SCAN PK_DEPT
- Rows from ID=4 are fed back to the parent ID=2 and these are fed back to it's parent ID=1.
- Because ID=1 is a NESTED LOOPS join, the first input is used to drive the join (this NESTED LOOPS ID=1 is illustrating the pre-fetch that was used by the NESTED LOOPS ID=2)
- ID=1 uses the rows to drive the lookup of it's second child ID=5: TABLE ACCESS BY INDEX ROWID| DEPT
- Rows from ID=5 are fed back to ID=1

- Rows from ID=1 are fed back to ID=0
- Rows from ID=0 are fed back to the client
- This process repeats until all the rows retched from ID=2 are exhausted

Execution order is 3,4,2,5,1,0

There are many ways of describing how to determine the operation in a plan, once familiar with a particular method it becomes second nature. One such description was to say that the rightmost-uppermost operation of an explain plan is executed first, but although this proved an intuitive description after some practice it is confusing to some readers. If in doubt consult the id and parent id hierarchy.

A good way of getting used to reading plans is to look at the plans in a tool like SQLT which outputs the execution order for you. For example, this query would appear as follows in SQLT:



SQL Text: [-]

```
select /*+ ordered USE_NL(dept) */ ename,dname
  from emp, dept
 where emp.deptno=dept.deptno
   and dept.dname in ('ACCOUNTING','RESEARCH','SALES','OPERATIONS')
```

SQL: [+]

ID	Exec Ord	Operation	Go To	More	Cost^2	Estim Card
0	6	SELECT STATEMENT			17	
1	5	NESTED LOOPS		[+]	18	
2	3	. NESTED LOOPS		[+]	17	14
3	1	.. TABLE ACCESS FULL EMP	[+]	[+]	3	14
4	2	.. INDEX UNIQUE SCAN PK_DEPT	[+]	[+]	0	1
5	4	. TABLE ACCESS BY INDEX ROWID DEPT	[+]	[+]	1	1

You can find more about SQLT in the following articles:

Document 215187.1 SQLT Diagnostic Tool
Document 1614107.1 SQLT Usage Instructions

Access Methods in detail

The Database SQL Tuning Guide provides a good overview of the various access methods here:

Oracle® Database SQL Tuning Guide
12c Release 1 (12.1)
E15858-15
Chapter 6 Optimizer Access Paths
http://docs.oracle.com/cd/E16655_01/server.121/e15858/tgsql_optop.htm#TGSQL228

The following section provides some other details and examples.

Full Table Scan (FTS)

In a FTS operation, the whole table is read up to the high water mark (HWM). The HWM marks the last block in the table that has ever had data written to it. If you have deleted all the rows then you will still read up to the HWM. Truncate resets the HWM back to the start of the table. FTS uses multi-block i/o to read the blocks from disk.

Buffers from FTS operations are placed on the Least Recently Used (LRU) end of the buffer cache so will be quickly aged out. In most cases FTS is not recommended for large tables unless you are reading >5-10% of it (or so) or you intend to run in parallel simply because there is likely to be a more economical way of accessing the data.

Example FTS explain plan:

```
SQL> set autot trace explain
SQL> select * from dual;

Execution Plan
----------------------------------------------------------
Plan hash value: 3543395131

--------------------------------------------------------------------------
| Id  | Operation         | Name | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |      |     1 |     2 |     2   (0)| 00:00:01 |
|   1 |  TABLE ACCESS FULL| DUAL |     1 |     2 |     2   (0)| 00:00:01 |
--------------------------------------------------------------------------
```

Index lookup

Data is accessed by looking up key values in an index and returning rowids. A rowid uniquely identifies an individual row in a particular data block. This block is read via single block i/o.

In this example an index is used to find the relevant row(s) and then the table is accessed to lookup the ename column (which is not included in the index):

```
SQL> select empno,ename from emp where empno=10;
```

```
Execution Plan
----------------------------------------------------------
Plan hash value: 4066871323

---------------------------------------------------------------------------------------
| Id  | Operation                   | Name   | Rows  | Bytes | Cost (%CPU)| Time     |
---------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |        |     1 |    10 |     1   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMP    |     1 |    10 |     1   (0)| 00:00:01 |
|*  2 |   INDEX UNIQUE SCAN         | PK_EMP |     1 |       |     0   (0)| 00:00:01 |
---------------------------------------------------------------------------------------

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

   2 - access("EMPNO"=10)
```

Notice the 'TABLE ACCESS BY ROWID' section. This indicates that the table data is not being accessed via a FTS operation but rather by a rowid lookup. In this case the rowid has been produced by looking up values in the index first. The index name in this case is PK_EMP and the index is being accessed by an 'INDEX UNIQUE SCAN' operation (explained below).

If all the required data resides in the index then a table lookup may be unnecessary and all you will see is an index access with no table access. In the following example all the columns (empno) are in the index. Notice that no table access takes place:

```
SQL> select empno from emp where empno=10;

Execution Plan
----------------------------------------------------------
Plan hash value: 4008335093

----------------------------------------------------------------------------
| Id  | Operation         | Name   | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |        |     1 |     4 |     0   (0)| 00:00:01 |
|*  1 |  INDEX UNIQUE SCAN| PK_EMP |     1 |     4 |     0   (0)| 00:00:01 |
----------------------------------------------------------------------------

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

   1 - access("EMPNO"=10)
```

Indexes are presorted, so sorting may be unecessary if the sort order required is the same as the index. For example:

```
select empno,ename from emp where empno > 7876 order by empno;

Execution Plan
----------------------------------------------------------
Plan hash value: 2449469783

---------------------------------------------------------------------------------------
| Id  | Operation                   | Name   | Rows  | Bytes | Cost (%CPU)| Time     |
---------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |        |     1 |    10 |     2   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMP    |     1 |    10 |     2   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN          | PK_EMP |     1 |       |     1   (0)| 00:00:01 |
---------------------------------------------------------------------------------------

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

   2 - access("EMPNO">7876)
```

Because the data in the index is already stored in sorted order, the rows are returned in sorted order of the index hence a sort is unnecessary assuming that the columns in the index are the same as in the order by. If we force an access path that does not use the index (e.g. FTS) then we will have to sort the data. For example:

```
 select /*+ Full(emp) */ empno,ename from emp where empno > 7876 order by empno;


Execution Plan
----------------------------------------------------------
Plan hash value: 4060621227

---------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
---------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     1 |    10 |     3   (0)| 00:00:01 |
|   1 |  SORT ORDER BY     |      |     1 |    10 |     3   (0)| 00:00:01 |
|*  2 |   TABLE ACCESS FULL| EMP  |     1 |    10 |     3   (0)| 00:00:01 |
---------------------------------------------------------------------------

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

   2 - access("EMPNO">7876)
```

Because we have forced a FTS the data is unsorted and so we must sort the data after it has been retrieved, hence the SORT ORDER BY step in the plan.

The following methods of index lookup are available:

- index unique scan
- index range scan
- index full scan
- index fast full scan
- index skip scan

Index unique scan

This is a method for looking up a single key value via a unique index and it always returns a single value. You must supply AT LEAST the leading column of the index to access data via the index (however this may return > 1 row as the uniqueness will not be guaranteed). For example:

```
SQL> select empno,ename from emp where empno=10;

Execution Plan
----------------------------------------------------------
Plan hash value: 4066871323

--------------------------------------------------------------------------------------
| Id  | Operation                   | Name   | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |        |     1 |    10 |     1   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMP    |     1 |    10 |     1   (0)| 00:00:01 |
|*  2 |   INDEX UNIQUE SCAN         | PK_EMP |     1 |       |     0   (0)| 00:00:01 |
--------------------------------------------------------------------------------------

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

   2 - access("EMPNO"=10)
```

Index range scan

Index range scan is a method for accessing a range values of a particular column. AT LEAST the leading column of the index must be supplied to access data via the index. It can be used for range operations (e.g. > < >= <= between) or where the data to be returned is not unique. For example:

```
SQL> select empno,ename from emp where empno > 7876 order by empno;

Execution Plan
----------------------------------------------------------
Plan hash value: 2449469783

--------------------------------------------------------------------------------------
| Id  | Operation                   | Name   | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |        |     1 |    10 |     2   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMP    |     1 |    10 |     2   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN          | PK_EMP |     1 |       |     1   (0)| 00:00:01 |
--------------------------------------------------------------------------------------

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

   2 - access("EMPNO">7876)
```

A non-unique index may return multiple values for the predicate mgr = 5 and will use an index range scan. For example:

```
SQL> create index emp_mgr on emp(mgr);
SQL> select mgr from emp where mgr = 5;

Execution Plan
----------------------------------------------------------
Plan hash value: 1542557660

----------------------------------------------------------------------------
| Id  | Operation        | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------
|   0 | SELECT STATEMENT |         |     1 |     4 |     1   (0)| 00:00:01 |
|*  1 |  INDEX RANGE SCAN| EMP_MGR |     1 |     4 |     1   (0)| 00:00:01 |
----------------------------------------------------------------------------

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

   1 - access("MGR"=5)
```

Index Full Scan

In certain circumstances it is possible for the whole index to be scanned as opposed to a range scan (i.e. where no constraining predicates are provided for a table). Full index scans are chosen when statistics indicate that it is going to be more efficient than a Full table scan and a sort. The index blocks are scanned one by one, not using multi-block I/O (like a FTS or Index FFS).

For example we may do a Full index scan when we do an unbounded scan of an index and want the data to be ordered in the index order. The optimizer may decide that selecting all the information from the index and not sorting is more efficient than doing a FTS or a Fast Full Index Scan and then sorting.

Index full scans are selected according to the criteria here:

> Oracle® Database SQL Tuning Guide
> 12c Release 1 (12.1)
> E15858-15
> Chapter 6 Optimizer Access Paths
> Section 6.3.4.1 When the Optimizer Considers Index Full Scans
> http://docs.oracle.com/cd/E16655_01/server.121/e15858/tgsql_optop.htm#TGSQL95166

An Index full scan will perform single block i/o's and so it may prove to be inefficient. In the following example, Index BE_IX is a concatenated index on emp (empno,ename). A select with no predictes results in an index full scan since it can satisfy the whole query without need to visit the table:

```
SQL> create index E_CIX on emp (empno,ename);

Index created.

SQL> select empno,ename from emp order by empno,ename;

Execution Plan
----------------------------------------------------------
Plan hash value: 2418964722

--------------------------------------------------------------------------
| Id  | Operation        | Name  | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------
|   0 | SELECT STATEMENT |       |    14 |   140 |     1   (0)| 00:00:01 |
|   1 |  INDEX FULL SCAN | E_CIX |    14 |   140 |     1   (0)| 00:00:01 |
--------------------------------------------------------------------------
```

Index Full Scan can also be used to access second column of concatenated indexes because the whole index is being retrieved as compared to a range scan which may not retrieve all the blocks.

```
SQL> select ename from emp;

Execution Plan
----------------------------------------------------------
Plan hash value: 2418964722

--------------------------------------------------------------------------
| Id  | Operation        | Name  | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------
|   0 | SELECT STATEMENT |       |    14 |    84 |     1   (0)| 00:00:01 |
|   1 |  INDEX FULL SCAN | E_CIX |    14 |    84 |     1   (0)| 00:00:01 |
--------------------------------------------------------------------------
```

Index Fast Full Scan (Index FFS)

An Index Fast Full Scan (Index FFS) scans all the blocks in the index using multiblock I/O. This means that the rows are not necessarily returned in sorted order. Index FFS may be hinted using INDEX_FFS hint and can be executed in parallel. It can also be used to access second column of concatenated indexes because the whole index is being retrieved as compared to a range scan which may not retrieve all the blocks.

Note that INDEX FAST FULL SCAN is the mechanism behind fast index create and recreate.

We can use the E_CIX concatenated index to illustrate an Index FFS:

```
SQL> select /*+ Index_FFS(emp) */ empno,ename from emp;

Execution Plan
----------------------------------------------------------
Plan hash value: 2100043038

------------------------------------------------------------------------------
| Id  | Operation            | Name  | Rows  | Bytes | Cost (%CPU)| Time     |
------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |       |    14 |   140 |     2   (0)| 00:00:01 |
|   1 |  INDEX FAST FULL SCAN| E_CIX |    14 |   140 |     2   (0)| 00:00:01 |
------------------------------------------------------------------------------
```

Index FFS can also be used to access second column of concatenated indexes because the whole index is being retrieved as compared to a range scan which may not retrieve all the blocks.

```
SQL> select /*+ Index_FFS(emp) */ ename from emp;

Execution Plan
----------------------------------------------------------
Plan hash value: 2100043038

------------------------------------------------------------------------------
| Id  | Operation            | Name  | Rows  | Bytes | Cost (%CPU)| Time     |
------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |       |    14 |    84 |     2   (0)| 00:00:01 |
|   1 |  INDEX FAST FULL SCAN| E_CIX |    14 |    84 |     2   (0)| 00:00:01 |
------------------------------------------------------------------------------
```

Index skip scan

Index skip scan finds rows even if the column is not the leading column of a concatenated index. It skips the first column(s) during the search. The next example checks ename='SMITH' for each index key even though ename is not the leading column of the index. The leading column (empno) is skipped.

```
SQL>  select /*+ index_ss(emp i_emp)*/ job from emp where ename='SMITH';

Execution Plan
----------------------------------------------------------
Plan hash value: 112616935

--------------------------------------------------------------------------------------------
| Id  | Operation                           | Name  | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |       |     1 |    14 |     2   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID BATCHED| EMP   |     1 |    14 |     2   (0)| 00:00:01 |
|*  2 |   INDEX SKIP SCAN                   | E_CIX |     1 |       |     1   (0)| 00:00:01 |
--------------------------------------------------------------------------------------------

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

```
   2 - access("ENAME"='SMITH')
       filter("ENAME"='SMITH')
```

Rowid

This is the quickest access method available to retrieve a single row. Oracle retrieves the specified block and extracts the rows it is interested in.
Most frequently seen in explain plans as Table access by Rowid :

```
SQL>  select * from dept where rowid = ':x';

Execution Plan
----------------------------------------------------------
Plan hash value: 1749648863

-----------------------------------------------------------------------------------
| Id  | Operation                  | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------
|   0 | SELECT STATEMENT           |      |     1 |    20 |     1   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS BY USER ROWID| DEPT |     1 |    20 |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------

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

   1 - access(CHARTOROWID(':x'))
```

In this case, because the user has supplied the rowid, the plan shows TABLE ACCESS BY USER ROWID. If the table is accessed by rowid following an index lookup then that looks like the following:

```
SQL> select empno,ename from emp where empno=10;

Execution Plan
----------------------------------------------------------
Plan hash value: 4066871323

--------------------------------------------------------------------------------------
| Id  | Operation                   | Name   | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |        |     1 |    10 |     1   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMP    |     1 |    10 |     1   (0)| 00:00:01 |
|*  2 |   INDEX UNIQUE SCAN         | PK_EMP |     1 |       |     0   (0)| 00:00:01 |
--------------------------------------------------------------------------------------

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

   2 - access("EMPNO"=10)
```

TABLE ACCESS BY INDEX ROWID indicates that the rowid comes from the index access in the INDEX UNIQUE SCAN step with ID=2.

Joins

A Join is a predicate that attempts to combine 2 row sources. We only ever join 2 row sources together. At a given time only one Join Step is performed even though underlying row sources may have been accessed in parallel. Once 2 row sources have been combined the resultant row source can start to be joined to other row sources. Note that some join methods (such as nested loops) allow a batch of fetched rows to be returned to the higher levels before fetching the next batch. The join order for a query is the order in which joins are performed and makes a significant difference to the way in which the query is executed. By accessing particular row sources first, certain predicates may be filled earlier than they would be if other join orders were taken and this may prevent certain access paths from being taken.

For example, suppose there is a concatenated index on table A (a.col1,a.col2) with a.col1 as the leading column.

Consider the following query:

```
select A.col4
from   A,B,C
where  B.col3 = 10
and    A.col1 = B.col1
and    A.col2 = C.col2
and    C.col3 = 5
```

We could represent the joins present in the query using the following schematic representation:

```
                B    ---   A  ---    C

Predicates: col3=10                col3=5
```

There are really only 2 ways we can drive the query: via B.col3 or C.col3.
We would have to do a Full scan of A to be able to drive off A since there are no limiting predicates which is unlikely to be efficient with large tables. If we drive off table B, using predicate B.col3=10 (as a filter or lookup key), then we will retrieve the value for B.col1 and join to A.col1. Because we have now filled the leading column of the concatenated index on table A we can use this index to give us values for A.col2 and join to A. However, if we drive off table C, then we only get a value for a.col2 and since this is a trailing column of a concatenated index and the leading column has not been supplied at this point, we cannot use the index on a to lookup the data in A. This means that it is likely that the best join order will be B A C.

The CBO will obviously use costs to establish whether the individual access paths are a good idea or not.

If the CBO does not choose the join order we want, then one option is to hint the desired order by changing the from clause to read:

```
from B,A,C
```

and using the /*+ ordered */ hint. The resultant query would be:

```
select /*+ ordered */ A.col4
from   B,A,C
where  B.col3 = 10
and    A.col1 = B.col1
and    A.col2 = C.col2
and    C.col3 = 5
```

There are 3 main join types: Nested Loops (NL), Hash Join and Sort Merge Join (SMJ).

Nested Loops

Fetches the first batch of rows from row source 1, then probes row source 2 once for each row returned from row source 1

```
Row source Row 1 ------> Nested Loops Join -- Probe ->       Row source 2
Row source Row 2 ------> Nested Loops Join -- Probe ->       Row source 2
Row source Row 3 ------> Nested Loops Join -- Probe ->       Row source 2
...
```

The first table in a nested loops join (in this case Row source 1) is known as the outer table (or the driving table)
The probed table (in this case Row source 2) is known as the inner table
Accessing row source 2 is known as probing the inner table.

For nested loops to be efficient it is important that the first row source returns as few rows as possible as this directly controls the number of probes of the second row source. Also it helps if the access method for row source 2 is efficient as this operation is being repeated once for every row returned by row source 1.

```
SQL> select /*+ ordered USE_NL(d) */ ename,dname
  from emp e, dept d
 where e.deptno=d.deptno
   and d.dname in ('ACCOUNTING','RESEARCH','SALES','OPERATIONS');

Execution Plan
----------------------------------------------------------
Plan hash value: 196120631

---------------------------------------------------------------------------------------
| Id  | Operation                    | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
---------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |         |    14 |   308 |    17   (0)| 00:00:01 |
|   1 |  NESTED LOOPS                |         |       |       |            |          |
|   2 |   NESTED LOOPS               |         |    14 |   308 |    17   (0)| 00:00:01 |
|   3 |    TABLE ACCESS FULL         | EMP     |    14 |   126 |     3   (0)| 00:00:01 |
|*  4 |    INDEX UNIQUE SCAN         | PK_DEPT |     1 |       |     0   (0)| 00:00:01 |
|*  5 |   TABLE ACCESS BY INDEX ROWID| DEPT    |     1 |    13 |     1   (0)| 00:00:01 |
---------------------------------------------------------------------------------------

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

   4 - access("E"."DEPTNO"="D"."DEPTNO")
   5 - filter("D"."DNAME"='ACCOUNTING' OR "D"."DNAME"='OPERATIONS' OR
              "D"."DNAME"='RESEARCH' OR "D"."DNAME"='SALES')
```

Hash Join

Smallest row source is chosen and used to build a hash table and a bitmap. The second row source is hashed and checked against the hash table looking for joins. The bitmap is used as a quick lookup to check if rows are in the hash table and are especially useful when the hash table is too large to fit in memory.

```
SQL> select /*+ ordered USE_HASH(d) */ ename,dname
  from emp e, dept d
 where e.deptno=d.deptno
   and d.dname in ('ACCOUNTING','RESEARCH','SALES','OPERATIONS');

Execution Plan
----------------------------------------------------------
Plan hash value: 2285423260

---------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
---------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |    14 |   308 |     6   (0)| 00:00:01 |
|*  1 |  HASH JOIN         |      |    14 |   308 |     6   (0)| 00:00:01 |
|   2 |   TABLE ACCESS FULL| EMP  |    14 |   126 |     3   (0)| 00:00:01 |
|*  3 |   TABLE ACCESS FULL| DEPT |     4 |    52 |     3   (0)| 00:00:01 |
---------------------------------------------------------------------------

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

   1 - access("E"."DEPTNO"="D"."DEPTNO")
   3 - filter("D"."DNAME"='ACCOUNTING' OR "D"."DNAME"='OPERATIONS' OR
              "D"."DNAME"='RESEARCH' OR "D"."DNAME"='SALES')
```

Sort Merge Join

Rows are produced by Row Source 1 and are then sorted. Rows from Row Source 2 are then produced and sorted by the same sort key as Row Source 1. Row Source 1 and 2 are NOT accessed concurrently Sorted rows from both sides are then merged together (joined)

```
            MERGE
            /    \
         SORT    SORT
          /        \
    Row Source 1  Row Source 2
```

If the row sources are already (known to be) sorted then the sort operation is unnecessary as long as both 'sides' are sorted using the same key.
Presorted row sources include indexed columns and row sources that have already been sorted in earlier steps. Although the merge of the 2 row sources is handled serially, the row sources could be accessed in parallel.

```
SQL> SELECT d.dname,e.ename
FROM   dept d,emp e
WHERE  e.deptno = d.deptno
and d.dname in ('ACCOUNTING','RESEARCH','SALES','OPERATIONS');

Execution Plan
----------------------------------------------------------
Plan hash value: 2865896559

----------------------------------------------------------------------------------------
| Id  | Operation                    | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |         |    14 |   308 |     5   (0)| 00:00:01 |
|   1 |  MERGE JOIN                  |         |    14 |   308 |     5   (0)| 00:00:01 |
|*  2 |   TABLE ACCESS BY INDEX ROWID| DEPT    |     4 |    52 |     2   (0)| 00:00:01 |
|   3 |    INDEX FULL SCAN           | PK_DEPT |     4 |       |     1   (0)| 00:00:01 |
|*  4 |   SORT JOIN                  |         |    14 |   126 |     3   (0)| 00:00:01 |
|   5 |    TABLE ACCESS FULL         | EMP     |    14 |   126 |     3   (0)| 00:00:01 |
----------------------------------------------------------------------------------------

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

   2 - filter("D"."DNAME"='ACCOUNTING' OR "D"."DNAME"='OPERATIONS' OR
              "D"."DNAME"='RESEARCH' OR "D"."DNAME"='SALES')
   4 - access("E"."DEPTNO"="D"."DEPTNO")
       filter("E"."DEPTNO"="D"."DEPTNO")
```

Sorting is an expensive operation, especially with large tables. Because of this, SMJ is often not a particularly efficient join method.

Cartesian Product

A Cartesian Product is done where they are no join conditions between 2 row sources and there is no alternative method of accessing the data. It is not really a join as such as there is no join! Often a common coding mistake is to leave a join out of a query which will result in a cartesian product, but it can be useful in some circumstances, especially where the table on the left hand side has a very small number of rows (typically 1) meaning that the right hand side only needs to be accessed once.

For example, notice that there is no join between the 2 tables in the following query:

```
SQL> select emp.deptno,dept.deptno
from emp,dept;

Execution Plan
----------------------------------------------------------
Plan hash value: 3359512422

--------------------------------------------------------------------------------
| Id  | Operation            | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |         |    56 |   336 |     8   (0)| 00:00:01 |
|   1 |  MERGE JOIN CARTESIAN|         |    56 |   336 |     8   (0)| 00:00:01 |
|   2 |   INDEX FULL SCAN    | PK_DEPT |     4 |    12 |     1   (0)| 00:00:01 |
|   3 |   BUFFER SORT        |         |    14 |    42 |     7   (0)| 00:00:01 |
|   4 |    TABLE ACCESS FULL | EMP     |    14 |    42 |     2   (0)| 00:00:01 |
--------------------------------------------------------------------------------
```

The CARTESIAN keyword indicate that we are doing a cartesian product.

Operations

This section explain certain operations that show up in explain plans, for example: sort, filter, view

Sorts

There are a number of different operations that promote sorts, for example: order by clauses, sort merge joins etc. In older releases, group by would be implemented by a sort but in later releases hash based sorting has tended top be more efficient. See:

Document 345048.1 "Group By" Clause Does Not Guarantee a Sort Without "Order By" Clause in 10g and Above

Note that if the row source is already appropriately sorted then no sorting is required. This is indicated by the NOSORT keyword:

```
SQL> select sum(sal) from emp where empno > 7876 group by empno order by empno;

Execution Plan
----------------------------------------------------------
Plan hash value: 3856122068
```

```
-------------------------------------------------------------------------------------
| Id  | Operation                    | Name   | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |        |     1 |     8 |     2   (0)| 00:00:01 |
|   1 |  SORT GROUP BY NOSORT        |        |     1 |     8 |     2   (0)| 00:00:01 |
|   2 |   TABLE ACCESS BY INDEX ROWID| EMP    |     1 |     8 |     2   (0)| 00:00:01 |
|*  3 |    INDEX RANGE SCAN          | PK_EMP |     1 |       |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------

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

   3 - access("EMPNO">7876)
```

In this case the group by operation simply groups the rows. It does not do the sort operation as the data is already in the correct order.

Sorts are expensive operations especially on large tables where the rows do not fit in memory and spill to disk. By default sort blocks are placed into the buffer cache. This may result in aging out of other blocks that may be reread by other processes.

Filter

The filter operation has a number of different meanings, for example:

- where one row source is filtering another
- functions such as min or max may introduce filter steps into query plans
- used to indicate partition elimination

In the following example the filter step acts similarly to a NL join except that it stops when it gets a row that does not match the criteria (i.e.like a bounded NL). The way the query works is that Step 2 executes first, followed by step 3, then the rows are filtered using the criteria in step 1. With larger row counts, filter operations tend not to be particularly efficient and so they have largely been replaced over time with query transformations opening up new access paths dependant on the exact query (for example subquery unnesting and anti/semi joins etc.) In this example the subquery unnesting has been disabled to show the filters:

```
SQL> ALTER SESSION SET "_unnest_subquery" = FALSE;
SQL> select * from dept d where not exists (select NULL from emp x where x.deptno = d.deptno and sal > 10000);

Execution Plan
----------------------------------------------------------
Plan hash value: 1421890032

---------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
---------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     3 |    60 |     9   (0)| 00:00:01 |
|*  1 |  FILTER            |      |       |       |            |          |
|   2 |   TABLE ACCESS FULL| DEPT |     4 |    80 |     3   (0)| 00:00:01 |
|*  3 |   TABLE ACCESS FULL| EMP  |     1 |     7 |     3   (0)| 00:00:01 |
---------------------------------------------------------------------------

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

   1 - filter( NOT EXISTS (SELECT 0 FROM "EMP" "X" WHERE "SAL">10000
              AND "X"."DEPTNO"=:B1))
   3 - filter("SAL">10000 AND "X"."DEPTNO"=:B1)
```

Views

When a view cannot be merged into the main query you will often see a projection view operation. You might see this because a subquery cannot be unnested any further or if there is a non-mergeable view included in the query due to restrictions. This indicates that the 'view' will be selected from directly as opposed to being broken down into joins on the base tables. A number of constructs make a view non mergeable and a few examples are shown below.

Inline view with aggregate function

In the following example the select contains an inline view which cannot be merged since the inline view tmp which contains an aggregate function cannot be merged into the main query:

```
SQL> SELECT ename, tmp.tot
FROM emp,   (SELECT empno,sum(empno) tot FROM emp GROUP BY empno) tmp
WHERE emp.empno = tmp.empno;

Execution Plan
----------------------------------------------------------
Plan hash value: 1414210221

---------------------------------------------------------------------------------
| Id  | Operation               | Name   | Rows  | Bytes | Cost (%CPU)| Time     |
---------------------------------------------------------------------------------
|   0 | SELECT STATEMENT        |        |    14 |   322 |     1   (0)| 00:00:01 |
|   1 |  NESTED LOOPS           |        |    14 |   322 |     1   (0)| 00:00:01 |
|   2 |   INDEX FULL SCAN       | E_CIX  |    14 |   140 |     1   (0)| 00:00:01 |
|   3 |   VIEW PUSHED PREDICATE |        |     1 |    13 |     0   (0)| 00:00:01 |
|*  4 |    FILTER               |        |       |       |            |          |
|   5 |     SORT AGGREGATE      |        |     1 |     4 |            |          |
|*  6 |      INDEX UNIQUE SCAN  | PK_EMP |     1 |     4 |     0   (0)| 00:00:01 |
---------------------------------------------------------------------------------

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

   4 - filter(COUNT(*)>0)
   6 - access("EMPNO"="EMP"."EMPNO")
   6 - access("EMPNO"="EMP"."EMPNO")
```

The explain plan shows this as a view step.

Subquery unnested but resultant view not merged:

As part of query transformation, subqueries can be unnested (re-written so that they are combined with base query). There are numerous and complex rules governing this activity which are beyond the scope of this article. However, if successful, a subquery may be unnested to produce an inline view which represents the subquery. This view is subject to view merging. If the view is non-mergeable then a VIEW keyword will appear in the plan.

```
SQL> SELECT ename
FROM emp
WHERE emp.deptno in (SELECT deptno x
                     FROM dept
                     WHERE dept.dname in ('ACCOUNTING','RESEARCH','SALES','OPERATIONS')
                     GROUP BY deptno)
Execution Plan
----------------------------------------------------------
Plan hash value: 2626410520

-----------------------------------------------------------------------------------------------
| Id  | Operation                              | Name     | Rows  | Bytes | Cost  (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                       |          |    14 |   308 |     5   (0)| 00:00:01 |
|*  1 |  HASH JOIN SEMI                        |          |    14 |   308 |     5   (0)| 00:00:01 |
|   2 |   TABLE ACCESS FULL                    | EMP      |    14 |   126 |     3   (0)| 00:00:01 |
|   3 |   VIEW                                 | VW_NSO_1 |     4 |    52 |     2   (0)| 00:00:01 |
|   4 |    HASH GROUP BY                       |          |     4 |    52 |     2   (0)| 00:00:01 |
|*  5 |     TABLE ACCESS BY INDEX ROWID BATCHED| DEPT     |     4 |    52 |     2   (0)| 00:00:01 |
|   6 |      INDEX FULL SCAN                   | PK_DEPT  |     4 |       |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------

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

   1 - access("EMP"."DEPTNO"="X")
   5 - filter("DEPT"."DNAME"='ACCOUNTING' OR "DEPT"."DNAME"='OPERATIONS' OR
              "DEPT"."DNAME"='RESEARCH' OR "DEPT"."DNAME"='SALES')
```

In this example, the subquery is unnested but the resultant inline view cannot be merged due to the 'group by'. This unmergeable view is given the system generated name of "VW_NSO_1" and is joined with a semi join.

Note: In later versions, as with any code, certain inline views may be mergeable following code improvements or as new features are introduced.

Remote Queries

Queries that involve access to distributed systems indicate this by the use of the REMOTE keyword in the OPERATION column. Queries can be executed completely on a remote side or parts can be local and other parts remote. The Optimizer decides based upon costs based upon local and remote statistics how best to execute the query.

Fully Remote Query

The following query is wholly sent to the remote site. After the query has executed remotely , results are returned.

```
SQL> SELECT * FROM dept@loop_link;

Execution Plan
----------------------------------------------------------
Plan hash value: 1554282393

-----------------------------------------------------------------------------------------
| Id  | Operation              | Name | Rows  | Bytes | Cost (%CPU)| Time     | Inst   |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT REMOTE|      |     4 |    80 |     3   (0)| 00:00:01 |        |
|   1 |  TABLE ACCESS FULL     | DEPT |     4 |    80 |     3   (0)| 00:00:01 | V12101 |
-----------------------------------------------------------------------------------------

Note
-----
   - fully remote statement
```

Partially Remote Query

```
SQL> select a.dname,avg(b.sal),max(b.sal)
from dept@loop_link a, emp b
where a.deptno=b.deptno
group by a.dname
order by max(b.sal),avg(b.sal) desc;

Execution Plan
----------------------------------------------------------
Plan hash value: 3854887037

--------------------------------------------------------------------------------------------
| Id  | Operation            | Name | Rows  | Bytes | Cost (%CPU)| Time     | Inst   |IN-OUT|
--------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |      |     4 |   108 |     6   (0)| 00:00:01 |        |      |
|   1 |  SORT ORDER BY       |      |     4 |   108 |     6   (0)| 00:00:01 |        |      |
|   2 |   HASH GROUP BY      |      |     4 |   108 |     6   (0)| 00:00:01 |        |      |
|*  3 |    HASH JOIN         |      |    14 |   378 |     6   (0)| 00:00:01 |        |      |
|   4 |     REMOTE           | DEPT |     4 |    80 |     3   (0)| 00:00:01 | LOOP_~ | R->S |
|   5 |     TABLE ACCESS FULL| EMP  |    14 |    98 |     3   (0)| 00:00:01 |        |      |
-------------------------------------------------------------------------------------------

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

   3 - access("A"."DEPTNO"="B"."DEPTNO")

Remote SQL Information (identified by operation id):
----------------------------------------------------

   4 - SELECT "DEPTNO","DNAME" FROM "DEPT" "A" (accessing 'LOOP_LINK' )
```

The query executed on remote node and the location is shown in the output. (in the plan table this information is recorded in the OTHER and OTHER_NODE columns). For more details on remote queries see:

Document 33838.1 Determining the execution plan for a distributed query

Oracle® Database Administrator's Guide
12c Release 1 (12.1)
E17636-20
Chapter 33 Developing Applications for a Distributed Database System
Analyzing the Execution Plan
http://docs.oracle.com/cd/E16655_01/server.121/e17636/ds_appdev.htm#ADMIN12206

Bind Variables

Bind variables are recommended in most cases because they promote sharing of sql code. With bind variable peeking enabled, the optimizer can use the current bind variable value to determine the bind value that is submitted and will base the plan on that value, re-using it for future executions. For more details on bind variable peeking see:

Document 70075.1 Use of bind variables in queries
Document 387394.1 Query using Bind Variables is suddenly slow
Document 401068.1 Possible Poor Runtime Performance for Bind Variables when Compared with Literal Values

To check the execution plan of a query using bind variables in sqlplus, the following can be used:

```
SQL> variable x varchar2(18);

begin     :x := 'hello';
end;
/

SQL> select * from dept where rowid = ':x';

Execution Plan
----------------------------------------------------------
Plan hash value: 1749648863

----------------------------------------------------------------------------------
| Id  | Operation                  | Name | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------------
|   0 | SELECT STATEMENT           |      |     1 |    20 |     1   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS BY USER ROWID| DEPT |     1 |    20 |     1   (0)| 00:00:01 |
----------------------------------------------------------------------------------

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

   1 - access(CHARTOROWID(':x'))
```

How to obtain explain plans

Collection of explain plans is outlined in the following article:

Document 235530.1 How to Obtain a Formatted Explain Plan - Recommended Methods

The manual also provides useful material:

Oracle® Database SQL Tuning Guide
12c Release 1 (12.1)
E15858-15
Chapter 8 Generating and Displaying Execution Plans
http://docs.oracle.com/cd/E16655_01/server.121/e15858/tgsql_genplan.htm#TGSQL271

Chapter 9 Reading Execution Plans
http://docs.oracle.com/cd/E16655_01/server.121/e15858/tgsql_interp.htm#TGSQL94618

REFERENCES

NOTE:209197.1 - Using Statspack to Record Explain Plan Details
NOTE:41634.1 - TKProf Basic Overview
NOTE:235530.1 - * How to Obtain a Formatted Explain Plan - Recommended Methods
NOTE:236026.1 - Display Execution Plans with DBMS_XPLAN.DISPLAY
NOTE:179518.1 - Partition Pruning and Joins
NOTE:237287.1 - How To Verify Parallel Execution is Running
NOTE:33838.1 - Determining the execution plan for a distributed query

Related

Products

- Oracle Database Products > Oracle Database Suite > Oracle Database > Oracle Database - Enterprise Edition > RDBMS > Generic SQL Performance, SQL Execution, Query Optimizer
- More Applications > Industry Solutions > Communications > Oracle Communications MetaSolv Solution > Engineering > Connection Design

Keywords

SORT MERGE JOIN; NESTED LOOPS; QUERY EVALUATION PLAN

Back to Top