ORACLE DATABASE 12c

The Oracle Optimizer
Explain the Explain Plan

ORACLE WHITE PAPER | APRIL 2017

ORACLE

Table of Contents

Introduction

The purpose of the Oracle Optimizer is to determine the most efficient execution plan for your queries. It makes these decisions based on the statistical information it has about your data and by leveraging Oracle database features such as hash joins, parallel query, partitioning, etc. Still it is expected that the optimizer will generate sub-optimal plans for some SQL statements now and then. In cases where there is an alternative plan that performed better than the plan generated by the optimizer, the first step in diagnosing why the Optimizer picked the sub-optimal plan is to visually inspect both of the execution plans.

Examining the different aspects of an execution plan, from selectivity to parallel execution and understanding what information you should be gleaming from the plan can be overwhelming even for the most experienced DBA. This paper offers a detailed explanation about each aspect of the execution plan and an insight into what caused the CBO to make the decision it did.

The Execution Plan

An execution plan shows the detailed steps necessary to execute a SQL statement. These steps are expressed as a set of database operators that consume and produce rows. The order of the operators and their implementations is decided by the query optimizer using a combination of query transformations and physical optimization techniques. While the display is commonly shown in a tabular format, the plan is in fact tree-shaped. For example, consider the following query based on the SH schema (Sales History):

```
SELECT   prod_category, AVG(amount_sold)
FROM     sales s, products p
WHERE    p.prod_id = s.prod_id
GROUP BY prod_category;
```

The tabular representation of this query's plan is:

```
-------------------------------------------------------------------------------------------------
| Id  | Operation            | Name     | Rows  | Bytes | Cost (%CPU)| Time     | Pstart| Pstop |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |          |       |       |  1140 (100)|          |       |       |
|   1 |  HASH GROUP BY       |          |     4 |    80 |  1140  (45)| 00:00:14 |       |       |
|*  2 |   HASH JOIN          |          |   489K|  9555K|   792  (21)| 00:00:10 |       |       |
|   3 |    TABLE ACCESS FULL | PRODUCTS |   767 |  8437 |    10   (0)| 00:00:01 |       |       |
|   4 |    PARTITION RANGE ALL|         |   489K|  4300K|   741  (17)| 00:00:09 |     1 |    16 |
|   5 |     TABLE ACCESS FULL| SALES    |   489K|  4300K|   741  (17)| 00:00:09 |     1 |    16 |
-------------------------------------------------------------------------------------------------
```

Figure 1: Tabular shaped execution plan

While the tree-shaped representation of the plan is:

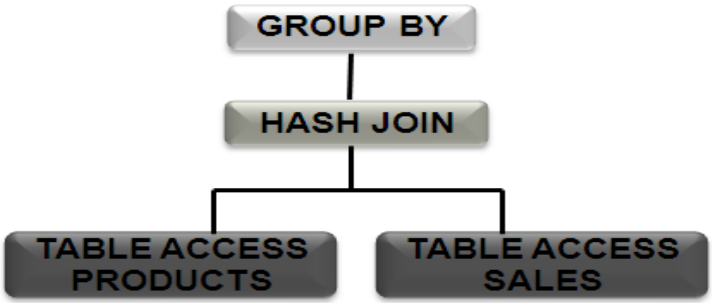


Figure 2: Tree shaped execution plan

The tabular representation is a top-down, left-to-right traversal of the execution tree. When you read a plan tree you should start from the bottom left and work across and then up. In the above example, begin by looking at the leaves of the tree. In this case the leaves of the tree are implemented using a full table scans of the PRODUCTS and the SALES tables. The rows produced by these table scans will be consumed by the join operator. Here the join operator is a hash-join (other alternatives include nested-loop or sort-merge join). Finally the group-by operator implemented here using hash (alternative would be sort) consumes rows produced by the join-operator, and return the final result set to the end user.

Displaying the Execution Plan

The two most common methods used to display the execution plan of a SQL statement are:

EXPLAIN PLAN command - This displays an execution plan for a SQL statement without actually executing the statement.

V$SQL_PLAN - A dynamic performance view that shows the execution plan for a SQL statement that has been compiled into a cursor and stored in the cursor cache.

Under certain conditions the plan shown when using EXPLAIN PLAN can be different from the plan shown using V$SQL_PLAN. For example, when the SQL statement contains bind variables the plan shown from using EXPLAIN PLAN ignores the bind variable values while the plan shown in V$SQL_PLAN takes the bind variable values into account in the plan generation process.

It is easy to display an execution plan using the DBMS_XPLAN package. This package provides several PL/SQL interfaces to display the plan from different sources:

- EXPLAIN PLAN command
- V$SQL_PLAN
- Automatic Workload Repository (AWR)
- SQL Tuning Set (STS)
- SQL Plan Baseline (SPM)

Using the EXPLAIN PLAN Command and the DBMS_XPLAN.DISPLAY Function

The following examples illustrate how to generate and display an execution plan for our original SQL statement using the different functions provided in the DBMS_XPLAN package.

```
SQL> EXPLAIN PLAN FOR
  2  Select prod_category, avg(amount_sold)
  3  From   sales s, products p
  4  Where  p.prod_id = s.prod_id
  5  Group by prod_category;

Explained.

SQL>
SQL> Select plan_table_output
  2  From table(dbms_xplan.display('plan_table',null,'basic'));

PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------
Plan hash value: 504757596

-----------------------------------------------
| Id  | Operation                     | Name     |
-----------------------------------------------
|   0 | SELECT STATEMENT              |          |
|   1 |  HASH GROUP BY                |          |
|   2 |   HASH JOIN                   |          |
|   3 |    VIEW                       | VW_GBC_5 |
|   4 |     HASH GROUP BY             |          |
|   5 |      PARTITION RANGE ALL      |          |
|   6 |       TABLE ACCESS STORAGE FULL| SALES    |
|   7 |    TABLE ACCESS STORAGE FULL  | PRODUCTS |
-----------------------------------------------
```

Figure 3: EXPLAIN PLAN output using the BASIC output

The arguments for DBMS_XPLAN.DISPLAY are:

- plan table name (default 'PLAN_TABLE')
- statement_id (default null means the last statement inserted into the plan table)
- format, controls the amount of information displayed (default is 'TYPICAL')

To leverage the explain plan functionality you need the appropriate privileges to run the actual statement you are trying to explain. A default PLAN_TABLE exists for every user without the need to create it beforehand.

Using the DBMS_XPLAN.DISPLAY_CURSOR Function

Alternatively the execution plan for an executed SQL statement can be generated and displayed by using the DBMS_XPLAN.DISPLAY_CURSOR function. The following example shows the plan for the previously executed SQL statement in the session.

```
SQL> Select prod_category, avg(amount_sold)
  2  From   sales s, products p
  3  Where  p.prod_id = s.prod_id
  4  Group by prod_category;

PROD_CATEGORY                                     AVG(AMOUNT_SOLD)
------------------------------------------------- ----------------
Software/Other                                          34.1313997
Hardware                                                1344.50776
Electronics                                             125.551667
Photo                                                   188.064642
Peripherals and Accessories                             108.824588

SQL>
SQL> Select plan_table_output
  2  From table(dbms_xplan.display_cursor(NULL,NULL,'basic'));

PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------
EXPLAINED SQL STATEMENT:
------------------------
Select prod_category, avg(amount_sold) From   sales s, products p Where
 p.prod_id = s.prod_id Group by prod_category

Plan hash value: 504757596

-----------------------------------------------
| Id  | Operation                     | Name     |
-----------------------------------------------
|   0 | SELECT STATEMENT              |          |
|   1 |  HASH GROUP BY                |          |
|   2 |   HASH JOIN                   |          |
|   3 |    VIEW                       | VW_GBC_5 |
|   4 |     HASH GROUP BY             |          |
|   5 |      PARTITION RANGE ALL      |          |
|   6 |       TABLE ACCESS STORAGE FULL| SALES    |
|   7 |    TABLE ACCESS STORAGE FULL  | PRODUCTS |
-----------------------------------------------
```

Figure 4: Execution plan accessing the SQL cursor cache, using the basic format

The arguments accepted by DBMS_XPLAN.DISPLAY_CURSOR are:

- SQL ID (default null, means the last SQL statement executed in this session),
- child number (default 0),
- format, controls the amount of information displayed (default 'TYPICAL')

Besides the privileges to actually run the SQL statement, the executing user needs SELECT privilege on VSQL_PLAN, VSQL_PLAN_DETAIL and SELECT_CATALOG_ROLE.

Formatting the execution plan

The format parameter for the functions in the DBMS_XPLAN package is highly customizable and can display as little (high-level) or as much (low-level) details as required or desired in the plan output. There are three pre-defined formats available:

- **BASIC** The plan includes only the ID, operation, and the object name columns.
- **TYPICAL** Includes the information shown in BASIC plus additional optimizer-related internal information such as cost, cardinality estimates, etc. This information is shown for every operation in the plan and represents what the optimizer thinks is the operation cost, the number of rows produced, etc. It also shows the predicates evaluated by each operation. There are two types of predicates: ACCESS and FILTER. The ACCESS predicates for an index are used to fetch the relevant blocks by applying search criteria to the appropriate columns. The FILTER predicates are evaluated after the blocks have been fetched.
- **ALL** Includes the information shown in TYPICAL plus the lists of expressions (columns) produced by every operation, the hint alias and query block names where the operation belongs (the outline information). The last two pieces of information can be used as arguments to add hints to the statement.

The low-level options allow the inclusion or exclusion of fine details, such as predicates and cost. The example in Figure 5 displays the basic execution plan and includes information on any predicates as well as the Optimizer Cost column.

```
SELECT plan_table_output
FROM   TABLE(DBMS_XPLAN.DISPLAY('plan_table',null,'basic +predicate +cost'));
```

```
-----------------------------------------------------------
| Id  | Operation               | Name     | Cost (%CPU)|
-----------------------------------------------------------
|   0 | SELECT STATEMENT        |          |  1101  (44)|
|   1 |  HASH GROUP BY          |          |  1101  (44)|
|*  2 |   HASH JOIN             |          |  1100  (43)|
|   3 |    TABLE ACCESS FULL    | PRODUCTS |    10   (0)|
|   4 |    VIEW                 | VW_GBC_5 |  1089  (44)|
|   5 |     HASH GROUP BY       |          |  1089  (44)|
|   6 |      PARTITION RANGE ALL|          |   741  (17)|
|   7 |       TABLE ACCESS FULL | SALES    |   741  (17)|
-----------------------------------------------------------

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

   2 - access("P"."PROD_ID"="ITEM_1")
```

Figure 5: Customized BASIC plan output with selected options PREDICATE and COST

It is also possible to use the low level arguments to exclude information from the plan. Figure 6 shows a sample plan where the Optimizer Cost and the Bytes columns are excluded.

```
SELECT plan_table_output
FROM   TABLE(DBMS_XPLAN.DISPLAY('plan_table',null,'typical -cost -bytes'));
```

```
-------------------------------------------------------------------------------
| Id  | Operation               | Name     | Rows  | Time     | Pstart| Pstop |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT        |          |     4 | 00:00:14 |       |       |
|   1 |  HASH GROUP BY          |          |     4 | 00:00:14 |       |       |
|*  2 |   HASH JOIN             |          |   766 | 00:00:14 |       |       |
|   3 |    TABLE ACCESS FULL    | PRODUCTS |   767 | 00:00:01 |       |       |
|   4 |    VIEW                 | VW_GBC_5 |   766 | 00:00:14 |       |       |
|   5 |     HASH GROUP BY       |          |   766 | 00:00:14 |       |       |
|   6 |      PARTITION RANGE ALL|          |   489K| 00:00:09 |     1 |    16 |
|   7 |       TABLE ACCESS FULL | SALES    |   489K| 00:00:09 |     1 |    16 |
-------------------------------------------------------------------------------

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

   2 - access("P"."PROD_ID"="ITEM_1")
```

Figure 6: Customized TYPICAL plan with suppressed options COST and BYTES

The Note Section

In addition to the plan and the predicate information, the DBMS_XPLAN package displays additional information in the NOTE section, such as when dynamic sampling was used during query optimization or that star transformation was applied to the query. In the example below the table SALES does not have statistics, so the optimizer has used dynamic sampling during the query optimization, which is displayed in the plan using the +note' in the query:

```
SELECT plan_table_output
FROM   TABLE(DBMS_XPLAN.DISPLAY('plan_table',null,'basic +note'));
```

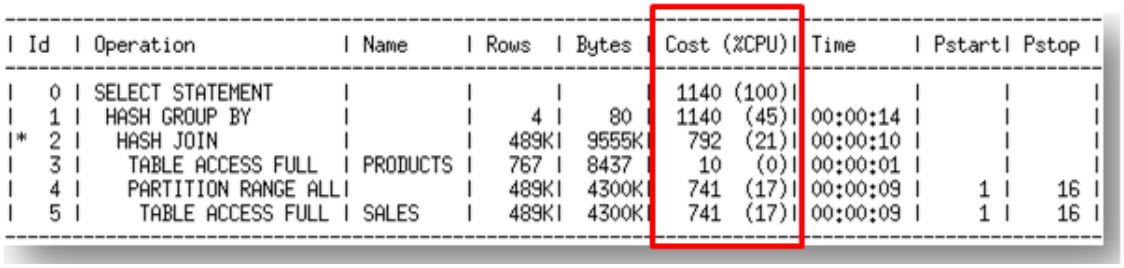
```
-----------------------------------------
| Id  | Operation             | Name     |
-----------------------------------------
|   0 | SELECT STATEMENT      |          |
|   1 |  HASH GROUP BY        |          |
|   2 |   HASH JOIN           |          |
|   3 |    TABLE ACCESS FULL  | PRODUCTS |
|   4 |    PARTITION RANGE ALL|          |
|   5 |     TABLE ACCESS FULL | SALES    |
-----------------------------------------

Note
-----
   - dynamic sampling used for this statement (level=2)
```

Figure 7: Basic plan output showing dynamic sampling was used for this statement

The note section is automatically displayed when the format option is set to either TYPICAL or ALL. More information on the DBMS_XPLAN package can be found in the Oracle® Database PL/SQL Packages and Types Reference guide.

What is Cost?

The Oracle Optimizer is a cost-based optimizer. The execution plan selected for a SQL statement is just one of the many alternative execution plans considered by the Optimizer. The Optimizer selects the execution plan with the lowest cost, where cost represents the estimated resource usage for that plan. The lower the cost the more efficient the plan is expected to be. The optimizer's cost model accounts for the IO, CPU, and network resources that will be used by the query.



Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time	Pstart	Pstop
0	SELECT STATEMENT				1140 (100)			
1	HASH GROUP BY		4	80	1140 (45)	00:00:14		
* 2	HASH JOIN		489K	9555K	792 (21)	00:00:10		
3	TABLE ACCESS FULL	PRODUCTS	767	8437	10 (0)	00:00:01		
4	PARTITION RANGE ALL		489K	4300K	741 (17)	00:00:09	1	16
5	TABLE ACCESS FULL	SALES	489K	4300K	741 (17)	00:00:09	1	16

Figure 8: Cost is found in the fifth column of the execution plan

The cost of the entire plan (indicated on line 0) and each individual operation is displayed in the execution plan. However, it is not something that can be tuned or changed. The cost is an internal unit and is only displayed to allow for plan comparisons.

Understanding the execution plan

In order to determine if you are looking at a good execution plan or not, you need to understand how the Optimizer determined the plan in the first place. You should also be able to look at the execution plan and assess if the Optimizer has made any mistake in its estimations or calculations, leading to a suboptimal plan. The components to assess are:

- **Cardinality**– Estimate of the number of rows coming out of each of the operations.
- **Access method** – The way in which the data is being accessed, via either a table scan or index access.
- **Join method** – The method (e.g., hash, sort-merge, etc.) used to join tables with each other.
- **Join type** – The type of join (e.g., outer, anti, semi, etc.).
- **Join order** – The order in which the tables are joined to each other.
- **Partition pruning** – Are only the necessary partitions being accessed to answer the query?
- **Parallel Execution** – In case of parallel execution, is each operation in the plan being conducted in parallel? Is the right data redistribution method being used?

Below is a detailed discussion on each of these components in the execution plan.

Cardinality

The cardinality is the estimated number of rows that will be returned by each operation. The Optimizer determines the cardinality for each operation based on a complex set of formulas that use both table and column level statistics as input (or the statistics derived by dynamic sampling). One of the simplest formulas is used when there is a single equality predicate in a single table query (with no histogram). In this case the Optimizer assumes a uniform distribution and calculates the cardinality for the query by dividing the total number of rows in the table by the number of distinct values in the column used in the where clause predicate.

Figure 9 shows a query running against the employees table in the HR schema, which has 107 rows:

```
SQL> SELECT employee_id, last_name, job_id
 2   FROM hr.employees
 3   WHERE job_id = 'AD_VP';

SQL> Select plan_table_output
 2   From table(dbms_xplan.display_cursor(null,null,'TYPICAL'));
```

```
-------------------------------------------------------------------------------------
| Id  | Operation                    | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |             |       |       |     2 (100)|          |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMPLOYEES   |     6 |   126 |     2   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN           | EMP_JOB_IX  |     6 |       |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------

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

   2 - access("JOB_ID"='AD_VP')
```

Figure 9: The CARDINALITY estimate is found in the Rows column of the execution plan

The job_id column has 19 distinct values so the optimizer predicted the cardinality for this statement to be 107/19 or 5.6 rows, which gets rounded up by DBMS_XPLAN to 6 rows.

It is import for the cardinality estimates to be as accurate as possible as they influence all aspects of the execution plan from the access method, to the join order. However, several factors can lead to incorrect cardinality estimates even when the basic table and column statistics are up to date. Some of these factors include:

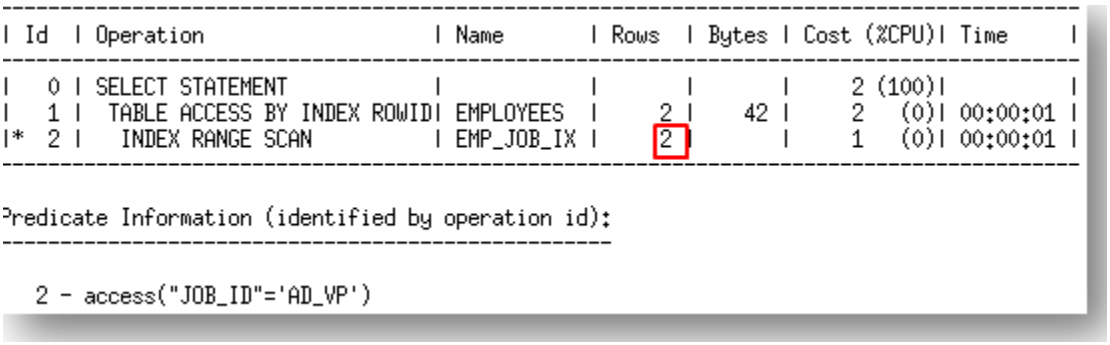
- Data skew
- Multiple single column predicates on a single table
- Function wrapped columns in the WHERE clause predicates
- Complex expressions

In the previous example there is a data skew in the EMPLOYEES table. There is not an even number of employees with each job_id. The actual number of rows in the employees table with a job_id of ‘AD_VP’ is only 2, which is 3 times less than the Optimizer originally estimated. In order to accurately reflect that data skew, a histogram is required on the JOB_ID column. The presence of a histogram changes the formula used by the Optimizer to determine the cardinality estimate.

By default Oracle automatically determines the columns that need histograms based on the column usage statistics and the presence of a data skew. If you need (or want) to create a histogram manually you can use the following command.

```
SQL> exec DBMS_STATS.GATHER_TABLE_STATS('HR', 'EMPLOYEES', method_opt=>'FOR COLUMNS SIZE 254 JOB_ID');
```

With a histogram on JOB_ID in place the optimizer estimates the correct number of rows will be returned from the sales table as seen in Figure 10.



```
-------------------------------------------------------------------------------------------
| Id  | Operation                    | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |             |       |       |     2 (100)|          |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMPLOYEES   |     2 |    42 |     2   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN           | EMP_JOB_IX  |     2 |       |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------

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

   2 - access("JOB_ID"='AD_VP')
```

Figure 10: Correct cardinality estimate with histogram present

Although having a more accurate cardinality estimate did not change the execution plan in this case it definitely can.

Determine the correct cardinality

To manually determine if the Optimizer has estimated the correct cardinality (or is in close proximity) you can use a simple `SELECT COUNT(*)` query for each tables used in the query and applying any WHERE clause predicates belonging to that table in the query. For the simple example used before

```
SQL> SELECT COUNT(*)
2     FROM hr.employees
3     WHERE job_id='AD_VP';


COUNT(*)
-------------
      2
```

Alternatively you can use the `GATHER_PLAN_STATISTICS` hint in the SQL statement to automatically collect more comprehensive runtime statistics. This hint records the actual cardinality (the number of rows returned) for each operation as the statement executes. This execution time (or run time) cardinality can then be displayed in the execution plan, using `DBMS_XPLAN.DISPLAY_CURSOR`, with the format parameter set to '`ALLSTATS LAST`'. An additional column called A-Rows, which stands for actual rows returned, will appear in the plan.

```
SQL> SELECT /*+ GATHER_PLAN_STATISTICS */ employee_id, last_name, job_id
  2  FROM  employees
  3  WHERE job_id='AD_VP';

SQL> SELECT plan_table_output
  2  FROM table(DBMS_XPLAN.DISPLAY_CURSOR (FORMAT=>'ALLSTATS LAST'));

-------------------------------------------------------------------------------------------------
| Id  | Operation                    | Name       | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |            |      1 |        |      2 |00:00:00.01 |       4 |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMPLOYEES  |      1 |      2 |      2 |00:00:00.01 |       4 |
|*  2 |   INDEX RANGE SCAN           | EMP_JOB_IX |      1 |      2 |      2 |00:00:00.01 |       2 |
-------------------------------------------------------------------------------------------------

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

   2 - access("JOB_ID"='AD_VP')
```

Compare actual rows returned by each operation (A-Rows) with the Optimizer estimate (E-Rows)

Figure 11: Runtime cardinality statistics are displayed in the A-Rows column

Note that using the `GATHER_PLAN_STATISTICS` hint has an impact on the execution time of a SQL statement, so you should use this only for analysis purposes. The `GATHER_PLAN_STATISTICS` hint, is not needed to display the A-Rows column when the init.ora parameter `STATISTICS_LEVEL` is set to `ALL`. The SQL*Monitoring functionality – either within Oracle Enterprise Manager or using the PL/SQL interface - will always display the A-Rows column information without any overhead for the SQL statement, as shown in Figure 12. Note that SQL*Monitoring is part of the 'Tuning and Diagnostics Pack and requires additional licensing.

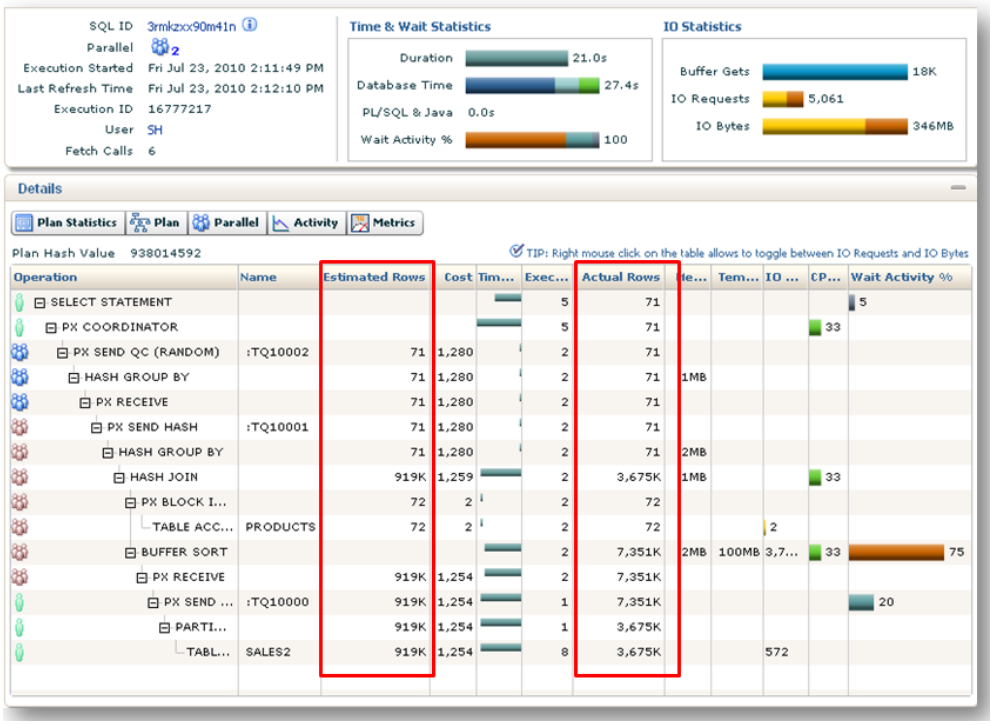
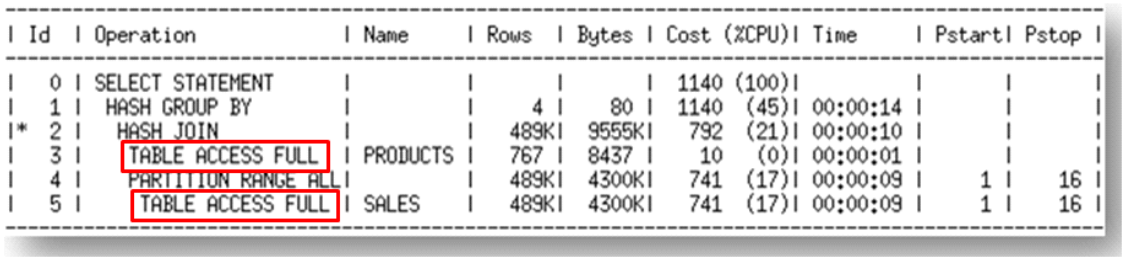


Figure 12: Sample execution plan as shown with SQL*Monitoring

Access Method

The access method - or access path - shows how the data will be accessed from each table (or index). The access method is shown in the operation field of the explain plan.



Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time	Pstart	Pstop
0	SELECT STATEMENT				1140 (100)			
1	HASH GROUP BY		4	80	1140 (45)	00:00:14		
* 2	HASH JOIN		489K	9555K	792 (21)	00:00:10		
3	TABLE ACCESS FULL	PRODUCTS	767	8437	10 (0)	00:00:01		
4	PARTITION RANGE ALL		489K	4300K	741 (17)	00:00:09	1	16
5	TABLE ACCESS FULL	SALES	489K	4300K	741 (17)	00:00:09	1	16

Figure 13: The access methods are shown in the Operations column of the plan

Oracle supports nine common access methods:

Full table scan - Reads all rows from a table and filters out those that do not meet the where clause predicates. A full table scan will use multi block IO (typically 1MB IOs). A full table scan is selected if a large portion of the rows in the table must be accessed, no indexes exist or the ones present can't be used or if the cost is the lowest. The decision to use a full table scan is also influenced by the following:

- Init.ora parameter db_multi_block_read_count
- Parallel degree
- Hints
- Lack of useable indexes
- Using an index costs more

Table access by ROWID – The rowid of a row specifies the data file, the data block within that file, and the location of the row within that block. Oracle first obtains the rowids either from a WHERE clause predicate or through an index scan of one or more of the table's indexes. Oracle then locates each selected row in the table based on its rowid and does a row-by-row access.

Index unique scan – Only one row will be returned from the scan of a unique index. It will be used when there is an equality predicate on a unique (B-tree) index or an index created as a result of a primary key constraint.

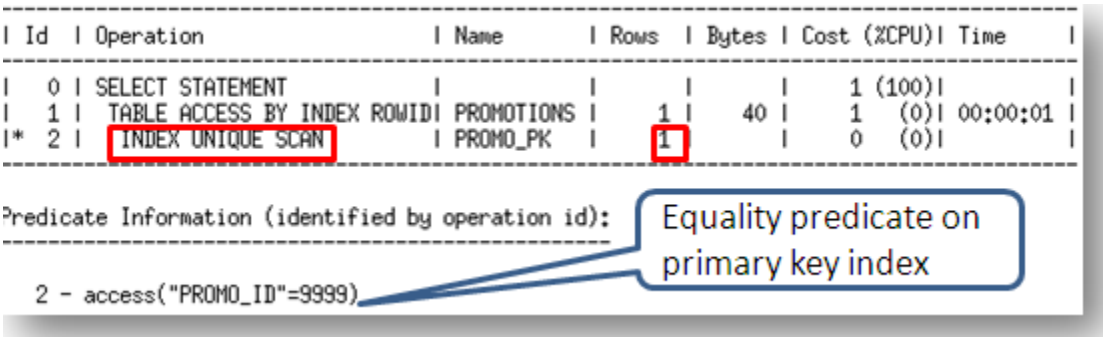


Figure 14: Plan using INDEX UNIQUE SCAN

Index range scan – Oracle accesses adjacent index entries and then uses the ROWID values in the index to retrieve the corresponding rows from the table. An index range scan can be bounded or unbounded. It will be used when a statement has an equality predicate on a non-unique index key, or a non-equality or range predicate on a unique index key. (=, <, >,LIKE if not on leading edge). Data is returned in the ascending order of index columns.

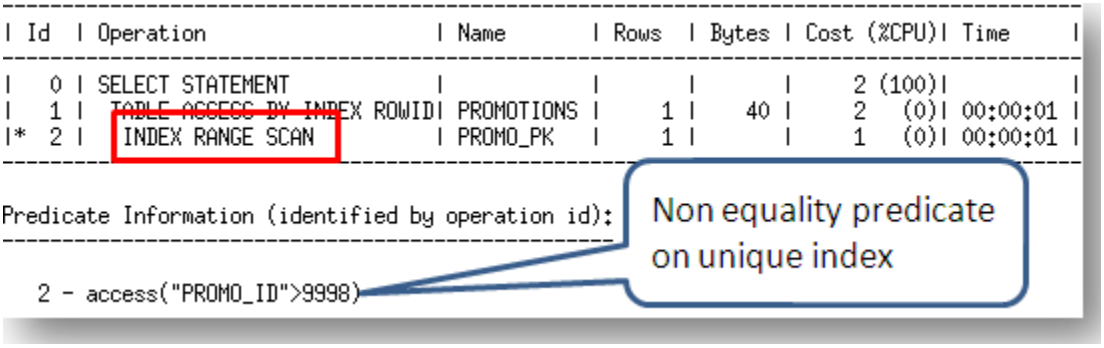


Figure 15: Plan using INDEX RANGE SCAN

Index range scan descending – Conceptually the same access as an index range scan, but it is used when an `ORDER BY .. DESCENDING` clause can be satisfied by an index.

Index skip scan - Normally, in order for an index to be used, the prefix of the index key (leading edge of the index) would be referenced in the query. However, if all the other columns in the index are referenced in the statement except the first column, Oracle can do an index skip scan, to skip the first column of the index and use the rest of it. This can be advantageous if there are few distinct values in the leading column of a concatenated index and many distinct values in the non-leading key of the index.

Full Index scan - A full index scan does not read every block in the index structure, contrary to what its name suggests. An index full scan processes all of the leaf blocks of an index, but only enough of the branch blocks to find the first leaf block. It is used when all of the columns necessary to satisfy the statement are in the index and it is cheaper than scanning the table. It uses single block IOs. It may be used in any of the following situations:

- An `ORDER BY` clause has all of the index columns in it and the order is the same as in the index (can also contain a subset of the columns in the index).
- The query requires a sort merge join and all of the columns referenced in the query are in the index.
- Order of the columns referenced in the query matches the order of the leading index columns.
- A `GROUP BY` clause is present in the query, and the columns in the `GROUP BY` clause are present in the index.

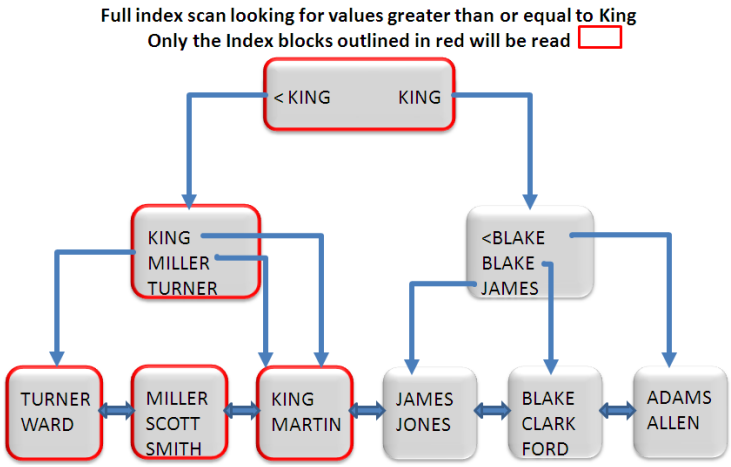


Figure 16: Processing of an INDEX FULL SCAN

Fast full index scan - This is an alternative to a full table scan when the index contains all the columns that are needed for the query, and at least one column in the index key has the NOT NULL constraint. It cannot be used to eliminate a sort operation, because the data access does not follow the index key. It will also read all of the blocks in the index using multiblock reads, unlike a full index scan.

Index join – This is a join of several indexes on the same table that collectively contain all of the columns that are referenced in the query from that table. If an index join is used, then no table access is needed, because all the relevant column values can be retrieved from the joined indexes. An index join cannot be used to eliminate a sort operation.

Bitmap Index – A bitmap index uses a set of bits for each key values and a mapping function that converts each bit position to a rowid. Oracle can efficiently merge bitmap indexes that correspond to several predicates in a WHERE clause, using Boolean operations to resolve AND and OR conditions.

If the access method you see in an execution plan is not what you expect, check the cardinality estimates for that object are correct and the join order allows the access method you desire.

Join Method

The join method describes how data from two data producing operators will be joined together. You can identify the join methods used in a SQL statement by looking in the operations column in the explain plan.

```
-------------------------------------------------------------------------------------------
| Id  | Operation            | Name     | Rows  | Bytes | Cost (%CPU)| Time     | Pstart| Pstop |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |          |       |       |   431 (100)|          |       |       |
|   1 |  HASH GROUP BY       |          |    71 |  2769 |   431  (11)| 00:00:01 |       |       |
|*  2 |   HASH JOIN          |          |   918K|    34M|   399   (4)| 00:00:01 |       |       |
|   3 |    TABLE ACCESS FULL | PRODUCTS |    72 |  2160 |     3   (0)| 00:00:01 |       |       |
|   4 |    PARTITION RANGE ALL|         |   918K|  8075K|   392   (3)| 00:00:01 |     1 |    28 |
|   5 |     TABLE ACCESS FULL | SALES    |   918K|  8075K|   392   (3)| 00:00:01 |     1 |    28 |
-------------------------------------------------------------------------------------------
```

Figure 17: Join Method is shown in the Operations column

Oracles offers several join methods and join types.

Join Methods

Hash Joins - Hash joins are used for joining large data sets. The optimizer uses the smaller of the two tables or data sources to build a hash table, based on the join key, in memory. It then scans the larger table, and performs the same hashing algorithm on the join column(s). It then probes the previously built hash table for each value and if they match, it returns a row.

Nested Loops joins - Nested loops joins are useful when small subsets of data are being joined and if there is an efficient way of accessing the second table (for example an index look up). For every row in the first table (the outer table), Oracle accesses all the rows in the second table (the inner table). Consider it like two embedded FOR loops. In Oracle Database 11g the internal implementation for nested loop joins changed to reduce overall latency for physical I/O so it is possible you will see two NESTED LOOPS joins in the operations column of the plan, where you previously only saw one on earlier versions of Oracle.

```
-------------------------------------------------------------------------------------------------------------
| Id  | Operation                      | Name           | Rows  | Bytes | Cost (%CPU)| Time     | Pstart| Pstop |
-------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT               |                |       |       | 16625 (100)|          |       |       |
|   1 |  HASH GROUP BY                 |                |    71 |  2769 | 16625   (1)| 00:00:01 |       |       |
|   2 |   NESTED LOOPS                 |                |       |       |            |          |       |       |
|   3 |    NESTED LOOPS                |                |  918K |   34M| 16593   (1)| 00:00:01 |       |       |
|   4 |     TABLE ACCESS FULL          | PRODUCTS       |    72 |  2160 |     3   (0)| 00:00:01 |       |       |
|   5 |     PARTITION RANGE ALL        |                |       |       |            |          |     1 |    28 |
|   6 |      BITMAP CONVERSION TO ROWIDS|               |       |       |            |          |       |       |
|*  7 |       BITMAP INDEX SINGLE VALUE| SALES_PROD_BIX |       |       |            |          |     1 |    28 |
|   8 |    TABLE ACCESS BY LOCAL INDEX ROWID| SALES     | 12762 |   112K| 16593   (1)| 00:00:01 |     1 |     1 |
-------------------------------------------------------------------------------------------------------------
```

Figure 18: Example plan output using NESTED LOOP

Sort Merge joins – Sort merge joins are useful when the join condition between two tables is an in-equality condition such as, <, <=, >, or >=. Sort merge joins can perform better than nested loop joins for large data sets. The join consists of two steps:

Sort join operation: Both the inputs are sorted on the join key.

Merge join operation: The sorted lists are merged together.

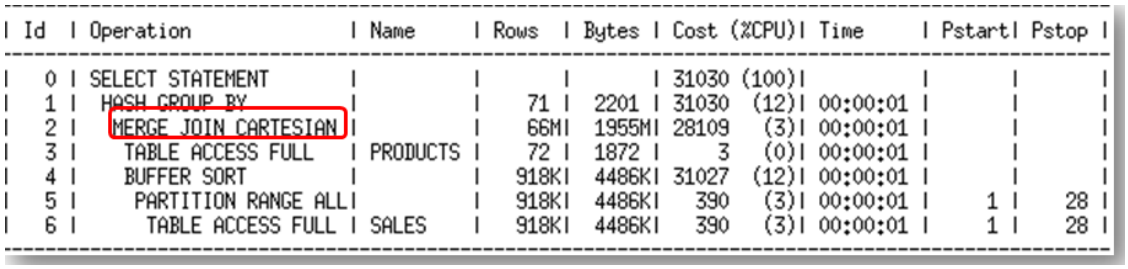
A sort merge join is more likely to be chosen if there is an index on one of the tables that will eliminate one of the sorts. In this example only the rows from the sales tables need to be sorted (ID 5), the rows from the products table are already sorted on the join column coming from the primary key index access (ID 4).

```
-------------------------------------------------------------------------------------------------------------
| Id  | Operation                      | Name        | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     | Pstart| Pstop |
-------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT               |             |       |       |       |  2628 (100)|          |       |       |
|   1 |  HASH GROUP BY                 |             |    71 |  2769 |       |  2628   (3)| 00:00:01 |       |       |
|   2 |   MERGE JOIN                   |             |  918K|   34M|       |  2596   (2)| 00:00:01 |       |       |
|   3 |    TABLE ACCESS BY INDEX ROWID| PRODUCTS    |    72 |  2160 |       |     2   (0)| 00:00:01 |       |       |
|   4 |     INDEX FULL SCAN            | PRODUCTS_PK |    72 |       |       |     1   (0)| 00:00:01 |       |       |
|*  5 |    SORT JOIN                   |             |  918K|  8075K|   35M|  2594   (2)| 00:00:01 |       |       |
|   6 |     PARTITION RANGE ALL        |             |  918K|  8075K|       |   392   (3)| 00:00:01 |     1 |    28 |
|   7 |      TABLE ACCESS FULL         | SALES       |  918K|  8075K|       |   392   (3)| 00:00:01 |     1 |    28 |
-------------------------------------------------------------------------------------------------------------
```

Figure 19: Example plan output using SORT MERGE JOIN

Beginning with Oracle Database 12c Release 2, *band joins* available. They make certain merge joins more efficient where there are BETWEEN predicates.

Cartesian join - The optimizer joins every row from one data source with every row from the other data source, creating a Cartesian product of the two sets. Typically this is only chosen if the tables involved are small or if one or more of the tables does not have a join conditions to any other table in the statement. Cartesian joins are not common, so it can be a sign of problem with the cardinality estimates, if it is selected for any other reason. Strictly speaking, a Cartesian product is not a join.



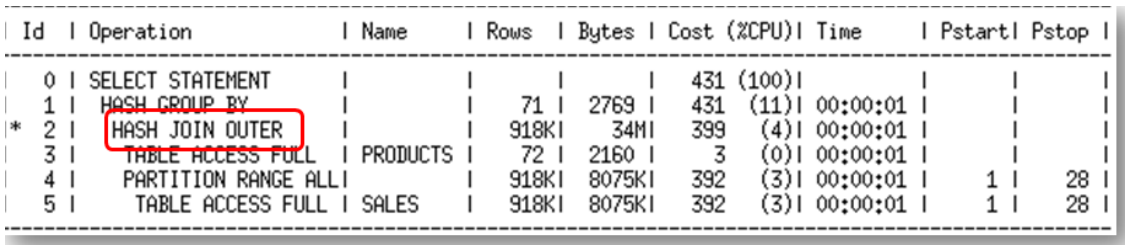
```
-------------------------------------------------------------------------------------------------
| Id  | Operation             | Name     | Rows  | Bytes | Cost (%CPU)| Time     | Pstart| Pstop |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |          |       |       | 31030 (100)|          |       |       |
|   1 |  HASH GROUP BY        |          |    71 |  2201 | 31030  (12)| 00:00:01 |       |       |
|   2 |   MERGE JOIN CARTESIAN|          |    66M|  1955M| 28109   (3)| 00:00:01 |       |       |
|   3 |    TABLE ACCESS FULL  | PRODUCTS |    72 |  1872 |     3   (0)| 00:00:01 |       |       |
|   4 |    BUFFER SORT        |          |   918K|  4486K| 31027  (12)| 00:00:01 |       |       |
|   5 |     PARTITION RANGE ALL|         |   918K|  4486K|   390   (3)| 00:00:01 |     1 |    28 |
|   6 |      TABLE ACCESS FULL | SALES   |   918K|  4486K|   390   (3)| 00:00:01 |     1 |    28 |
-------------------------------------------------------------------------------------------------
```

Figure 20: Example plan output using CARTESIAN JOIN

Join Types

Oracle offers several join types: inner join, (left) outer join, full outerjoin, anti join, semi join, grouped outer join, etc. Note that inner join is the most common type of join; hence the execution plan does not specify the key word "INNER'.

Outer Join - An outer join returns all rows that satisfy the join condition and also all of the rows from the table without the (+) for which no rows from the other table satisfy the join condition. For example, `T1.x = T2.x (+)`, here T1 is the left table whose non-joining rows will be retained. In the ANSI outer join syntax, it is the leading table whose non-join rows will be retained. The same example can be written in ANSI SQL as `T1 LEFT OUTER JOIN T2 ON (T1.x = T2.x);`



```
-------------------------------------------------------------------------------------------------
| Id  | Operation             | Name     | Rows  | Bytes | Cost (%CPU)| Time     | Pstart| Pstop |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |          |       |       |   431 (100)|          |       |       |
|   1 |  HASH GROUP BY        |          |    71 |  2769 |   431  (11)| 00:00:01 |       |       |
|*  2 |   HASH JOIN OUTER     |          |   918K|    34M|   399   (4)| 00:00:01 |       |       |
|   3 |    TABLE ACCESS FULL  | PRODUCTS |    72 |  2160 |     3   (0)| 00:00:01 |       |       |
|   4 |    PARTITION RANGE ALL|          |   918K|  8075K|   392   (3)| 00:00:01 |     1 |    28 |
|   5 |     TABLE ACCESS FULL | SALES    |   918K|  8075K|   392   (3)| 00:00:01 |     1 |    28 |
-------------------------------------------------------------------------------------------------
```

Figure 21: Example plan output using OUTER JOIN. Note a join type is always matched with one of the join methods; in this case a hash join

Join Order

The join order is the order in which the tables are joined together in a multi-table SQL statement. To determine the join order in an execution plan look at the indentation of the tables in the operation column. In Figure 22 below the `SALES` and `PRODUCTS` table are equally indented and both of them are more indented than the CUSTOMERS table. Therefore the `SALES` and `PRODUCTS` table will be joined first using a hash join and the result of that join will then be joined to the `CUSTOMERS` table.

```
-------------------------------------------------------------------------------------------------------
| Id  | Operation              | Name      | Rows  | Bytes | Cost (%CPU)| Time     | Pstart| Pstop |
-------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT       |           |       |       |   713 (100)|          |       |       |
|   1 |  HASH GROUP BY         |           |  2193 |   107K|   713   (8)| 00:00:01 |       |       |
|*  2 |   HASH JOIN            |           |   918K|    43M|   681   (3)| 00:00:01 |       |       |
|   3 |  3 TABLE ACCESS FULL   | CUSTOMERS | 55500 |   812K|   278   (1)| 00:00:01 |       |       |
|*  4 |    HASH JOIN           |           |   918K|    30M|   399   (4)| 00:00:01 |       |       |
|   5 |    1TABLE ACCESS FULL  | PRODUCTS  |    72 |  1512 |     3   (0)| 00:00:01 |       |       |
|   6 |     PARTITION RANGE ALL|           |   918K|    12M|   392   (3)| 00:00:01 |     1 |    28 |
|   7 |    2 TABLE ACCESS FULL | SALES     |   918K|    12M|   392   (3)| 00:00:01 |     1 |    28 |
-------------------------------------------------------------------------------------------------------
```

Figure 20: Example plan output highlighting the JOIN ORDER

In a more complex SQL statement it may not be so easy to determine the join order by looking at the indentations of the tables in the operations column. In these cases it might be easier to use the `FORMAT` parameter in the `DBMS_XPLAN` procedures to display the outline information for the plan, which will contain the join order. For example, to generate the outline information for the plan shown in Figure 22 the following format option of the DBMS_XPLAN can be used;

```
DBMS_XPLAN.DISPLAY_CURSOR(FORMAT=>'TYPICAL + OUTLINE'));
```

```
Outline Data
-------------

  /*+
      BEGIN_OUTLINE_DATA
      IGNORE_OPTIM_EMBEDDED_HINTS
      OPTIMIZER_FEATURES_ENABLE('11.2.0.2')
      DB_VERSION('11.2.0.2')
      ALL_ROWS
      OUTLINE_LEAF(@"SEL$1")
      FULL(@"SEL$1" "P"@"SEL$1")
      FULL(@"SEL$1" "S"@"SEL$1")
      FULL(@"SEL$1" "C"@"SEL$1")
      LEADING(@"SEL$1" "P"@"SEL$1" "S"@"SEL$1" "C"@"SEL$1")
      USE_HASH(@"SEL$1" "S"@"SEL$1")
      USE_HASH(@"SEL$1" "C"@"SEL$1")
      SWAP_JOIN_INPUTS(@"SEL$1" "C"@"SEL$1")
      USE_HASH_AGGREGATION(@"SEL$1")
      END_OUTLINE_DATA
  */
```

Figure 23: Outline for execution plan

In the outline information, look for the line that begins with the word `LEADING`. This line shows the join order for this query. In this example you see “P”, then “S”, then “C” referenced on this line; these three letters were the aliases

used for the three involved tables in the query. The P (PRODUCTS) table joins to the S (SALES) table and then to the C (CUSTOMERS) table.

The join order is determined based on cost, which is strongly influenced by the cardinality estimates and the access paths available. The Optimizer will also always adhere to some basic rules:

- Joins that result in at most one row always go first. The Optimizer can determine this based on UNIQUE and PRIMARY KEY constraints on the tables.
- When outer joins are used the row preserving table (table without the outer join operator) must come after the other table in the predicate (table with the outer join operator) to ensure all of the additional rows that don't satisfy the join condition can be added to the result set correctly.
- When a subquery has been converted into an antijoin or semijoin, the tables from the subquery must come after those tables in the outer query block to which they were connected or correlated. However, hash antijoins and semijoins are able to override this ordering condition under certain circumstances.
- If view merging is not possible all tables in the view will be joined before joining to the tables outside the view.

If the join order is not what you expect check the cardinality estimates for each of the objects and the access methods are correct.

Partitioning

Partitioning allows a table, index or index-organized table to be subdivided into smaller pieces. Each piece of the database object is called a Partition. Partition pruning or Partition elimination is the simplest means to improve performance using Partitioning. For example, if an application has an ORDERS table that contains a record of all orders for the last 2 years, and this table has been partitioned by day, a query requesting orders for a single week would only access seven partitions of the ORDERS table instead of 730 partitions (the entire table).

Partition pruning is visible in an execution plan in the PSTART and PSTOP columns. The PSTART column contains the number of the first partition that will be accessed and PSTOP column contains the number of the last partition that will be accessed[1]. In Figure 24 four partitions from SALES are accessed, namely partitions 9,10,11, and 12.

```
-------------------------------------------------------------------------------------------------
| Id  | Operation             | Name     | Rows  | Bytes | Cost (%CPU)| Time     | Pstart| Pstop |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |          |     1 |    12 |    62  (20)| 00:00:01 |       |       |
|   1 |  SORT AGGREGATE       |          |     1 |    12 |            |          |       |       |
|   2 |   PX COORDINATOR      |          |       |       |            |          |       |       |
|   3 |    PX SEND QC (RANDOM)| :TQ10000 |     1 |    12 |            |          |       |       |
|   4 |     SORT AGGREGATE    |          |     1 |    12 |            |          |       |       |
|   5 |      PX BLOCK ITERATOR|          |   138K|  1626K|    62  (20)| 00:00:01 |     9 |    12 |
|*  6 |       TABLE ACCESS FULL| SALES   |   138K|  1626K|    62  (20)| 00:00:01 |     9 |    12 |
-------------------------------------------------------------------------------------------------
```

Figure 24: Example plan output highlighting Partition pruning for a single-level partitioned table

1 Note that not necessarily all partitions between PSTART and PSTOP have to be accessed. More details about Partitioning and Partition pruning can be found on OTN on the Partitioning page

A simple select statement that was run against a table that is partitioned by day and sub-partitioned by hash on the CUST_ID column is shown in Figure 21. In this case a lot more numbers appear in the PSTART, PSTOP columns. What do these additional numbers mean?

```
-------------------------------------------------------------------------------------------------
| Id  | Operation                | Name    | Rows  | Bytes | Cost (%CPU)| Time     | Pstart| Pstop |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT         |         |       |       |     5 (100)|          |       |       |
|   1 |  SORT AGGREGATE          |         |     1 |    13 |            |          |       |       |
|   2 |   PARTITION RANGE SINGLE|         |     1 |    13 |     5   (0)| 00:00:01 |     5 |     5 |
|   3 |    PARTITION HASH SINGLE|         |     1 |    13 |     5   (0)| 00:00:01 |     2 |     2 |
|*  4 |     TABLE ACCESS FULL    | RHP_TAB |     1 |    13 |     5   (0)| 00:00:01 |    10 |    10 |
-------------------------------------------------------------------------------------------------

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

   4 - filter(("CUST_ID"=9255 AND "TIME_ID"=TO_DATE(' 2008-01-01 00:00:00', 'syyyy-mm-dd
              hh24:mi:ss')))
```

Figure 21: Example plan output highlighting Partition pruning for a composite partitioned table

When using composite partitioning, Oracle numbers each of the partitions from 1 to n (absolute partition numbers). For a table that is partitioned on just one level, these absolute numbers represent the actual physical segments on disk of the single-level partitioned table.

In the case of a composite partitioned table, however, a partition is a logical entity and not represented on disk. Each partition is subdivided into so-called sub-partitions. Each sub-partition within a partition is numbered from 1 to m (relative sub-partition number within a single partition). Finally all of the sub-partitions in a composite-partitioned table are given a global number 1 to (n X m) (absolute sub-partition numbers); these absolute numbers represent the actual physical segments on disk of the composite partitioned table.

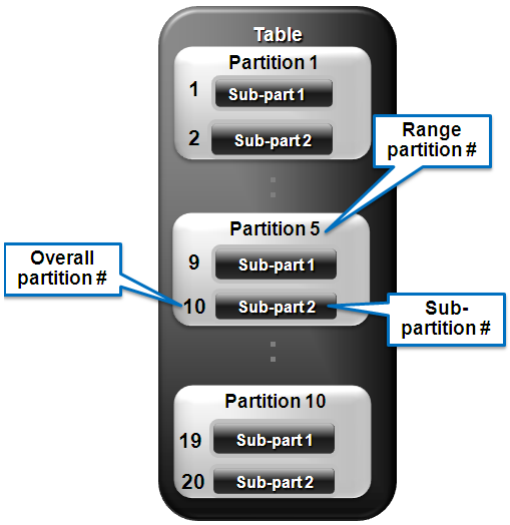


Figure 22: Numbering scheme for a partitioned table

So in the previous plan in Figure 21 the number 10 in PSTART and PSTOP column, on line 4 of the plan represents the global partitioning number representing the physical segments on disk. The number 5 in PSTART and PSTOP column, on line 2 of the plan represents the partition number; the number 2 in PSTART and PSTOP column, on line 3 of the plan, represents the relative sub-partition number within a partition.

There are cases when a word or letters appear in the PSTART and PSTOP columns instead of a number. For example you may see the word KEY appears in these columns. This indicates that it was not possible at parse time to identify, which partitions would be accessed by the query but the Optimizer believes that partition pruning will occur at execution time (dynamic pruning). This happens when there is an equality predicate on the partitioning key column that contains a function. For example TIME_ID = SYSDATE. Another situation where dynamic pruning can occur is when there is a join condition on the partitioning key column in the query and the table that is joined with the partitioned table is expected not to join with all partitions, for example because of a FILTER predicate. Partition pruning will occur at execution time. In the example in Figure27 below the where clause predicate is on the TIME table, which joins to the SALES table on the partition key TIME_ID. Partition pruning will happen at execution time after the WHERE clause predicate has been applied to the TIME table and the appropriate TIME_IDs have been selected.

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time	Pstart	Pstop
0	SELECT STATEMENT				41 (100)			
1	SORT AGGREGATE		1	29				
2	NESTED LOOPS		944	27376	41 (3)	00:00:01		
* 3	TABLE ACCESS STORAGE FULL	TIMES	2	32	13 (0)	00:00:01		
4	PARTITION RANGE ITERATOR		629	8177	14 (0)	00:00:01	KEY	KEY
* 5	TABLE ACCESS STORAGE FULL	SALES	629	8177	14 (0)	00:00:01	KEY	KEY

Figure27: Example plan output highlighting dynamic Partition pruning

If partition pruning does not occur as expected, check the predicates on the partition key column. Ensure that the predicates are using the same datatype as the partition key column. You can check this in the predicate information section under the plan. If the table is hash partitioned, partition pruning will only occur if the predicate on the partition key column is an equality or an in-list predicate. Also if the table has multi-column hash partitioning then partition pruning will only occur if there is a predicate on all columns used in the hash partitioning.

Parallel Execution

Parallel execution in the Oracle Database is based on the principles of a coordinator (often called the Query Coordinator or QC for short) and parallel server processes. The QC is the session that initiates the parallel SQL statement and the parallel server processes are the individual sessions that perform work in parallel. The QC distributes the work to the parallel server processes and may have to perform a minimal, mostly logistical, portion of the work that cannot be executed in parallel. For example a parallel query with a `SUM()` operation requires adding the individual sub-totals calculated by each parallel server processes.

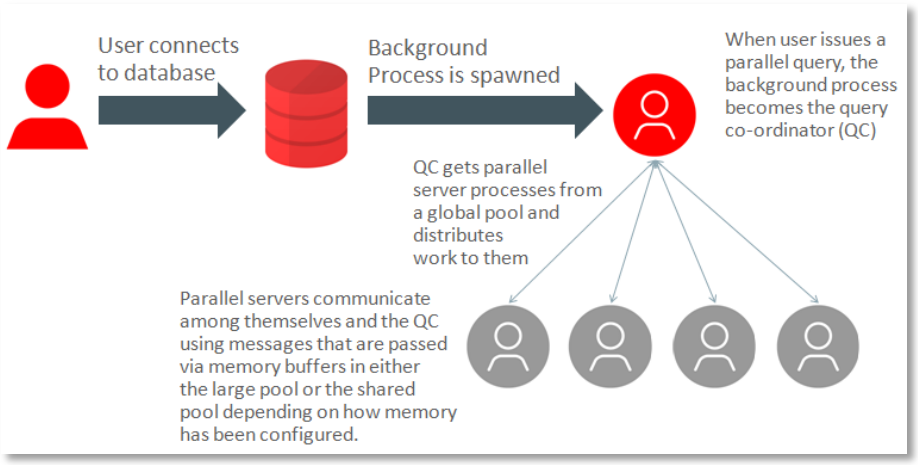


Figure 28: Concept of parallel execution in the Oracle database

The QC is easily identified in the parallel execution plan as it writes its name in the plan. You can see this on the line with ID#1 of the plan shown in Figure where you see the operation 'PX COORDINATOR'. All of the operations that appear above this line in the execution plan are done by the QC. Since this is a single process all of these operations are done serially. Typically you want to minimize the number of operations done by the QC. All of the operations done under the line 'PX COORDINATOR' are typically done by the parallel server processes.2

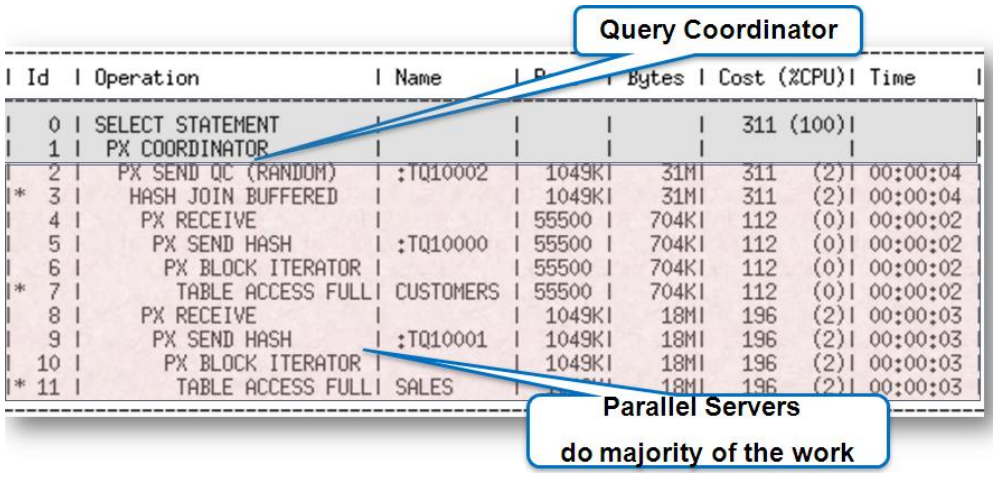


Figure 29: Example plan output highlighting the concepts of parallel execution

2 More details about Parallel Execution in Oracle can be found on OTN on the parallelism and scalability page

Granules

A granule is the smallest unit of work a parallel server processes can be given. The number of granules is normally much higher than the requested DOP in order to get an even distribution of work among parallel server processes. Each parallel server process will work exclusively on its own granule and when it finishes it will be given another granule to work on until all of the granules have been processed. The basic mechanism the Oracle Database uses to distribute work for parallel execution is block ranges on disk or **block-based granules**. In the execution plan you can see how the granules were distributed to the parallel server processes in the operations column on the line above the data access. The operation 'PX BLOCK ITERATOR ' shown on line 7 in the plan in Figure means the parallel server processes will iterate over the generated block range granules to complete the table scan.

```
-----------------------------------------------------------------------------------------------------------------------
| Id  | Operation                | Name     | Rows  | Bytes | Cost (%CPU)| Time     | Pstart| Pstop |    TQ  |IN-OUT| PQ Distrib |
-----------------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT         |          |    17 |   153 |   565 (100)| 00:00:07 |       |       |        |      |            |
|   1 |  PX COORDINATOR          |          |       |       |            |          |       |       |        |      |            |
|   2 |   PX SEND QC (RANDOM)    | :TQ10001 |    17 |   153 |   565 (100)| 00:00:07 |       |       |  Q1,01 | P->S | QC (RAND)  |
|   3 |    HASH GROUP BY         |          |    17 |   153 |   565 (100)| 00:00:07 |       |       |  Q1,01 | PCWP |            |
|   4 |     PX RECEIVE           |          |    17 |   153 |   565 (100)| 00:00:07 |       |       |  Q1,01 | PCWP |            |
|   5 |      PX SEND HASH        | :TQ10000 |    17 |   153 |   565 (100)| 00:00:07 |       |       |  Q1,00 | P->P | HASH       |
|   6 |       HASH GROUP BY      |          |    17 |   153 |   565 (100)| 00:00:07 |       |       |  Q1,00 | PCWP |            |
|   7 |        PX BLOCK ITERATOR |          |   10M |    85M|    60  (97)| 00:00:01 |     1 |    16 |  Q1,00 | PCWC |            |
|* 8 |         TABLE ACCESS FULL| SALES    |   10M |    85M|    60  (97)| 00:00:01 |     1 |    16 |  Q1,00 | PCWP |            |
-----------------------------------------------------------------------------------------------------------------------
```

Figure 30: Example plan output highlighting block granule processing

Although block-based granules are the most common approach, there are some operations that can benefit from leveraging the underlying data structure of a partitioned table. In these cases a partition becomes a granule of work. With **partition-based granules** one parallel server processes will perform the work for all of the data in a single partition. The Oracle Optimizer considers partition-based granules if the number of (sub)partitions accessed in the operation is at least equal to the DOP (and ideally much higher in case there is a skew in the sizes of the individual (sub)partitions). An example of partition-based granules can be seen in Figure31 line 6: 'PX PARTITION RANGE ALL' means that each parallel server process will work exclusively on one of the range partitions in the table.

```
-----------------------------------------------------------------------------------------------------------------------
| Id  | Operation                          | Name       | Rows  | Bytes | Cost (%CPU)| Time     | Pstart| Pstop |    TQ  |IN-OUT| PQ Distrib |
-----------------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                   |            |    17 |   153 |     2  (50)| 00:00:01 |       |       |        |      |            |
|   1 |  PX COORDINATOR                    |            |       |       |            |          |       |       |        |      |            |
|   2 |   PX SEND QC (RANDOM)              | :TQ10001   |    17 |   153 |     2  (50)| 00:00:01 |       |       |  Q1,01 | P->S | QC (RAND)  |
|   3 |    HASH GROUP BY                   |            |    17 |   153 |     2  (50)| 00:00:01 |       |       |  Q1,01 | PCWP |            |
|   4 |     PX RECEIVE                     |            |    26 |   234 |     1   (0)| 00:00:01 |       |       |  Q1,01 | PCWP |            |
|   5 |      PX SEND HASH                  | :TQ10000   |    26 |   234 |     1   (0)| 00:00:01 |       |       |  Q1,00 | P->P | HASH       |
|   6 |       PX PARTITION RANGE ALL       |            |    26 |   234 |     1   (0)| 00:00:01 |     1 |    16 |  Q1,00 | PCWC |            |
|   7 |        TABLE ACCESS BY LOCAL INDEX ROWID| SALES      |    26 |   234 |     1   (0)| 00:00:01 |     1 |    16 |  Q1,00 | PCWP |            |
|* 8 |         INDEX RANGE SCAN           | SALES_CUST |    26 |       |     0   (0)| 00:00:01 |     1 |    16 |  Q1,00 | PCWP |            |
-----------------------------------------------------------------------------------------------------------------------
```

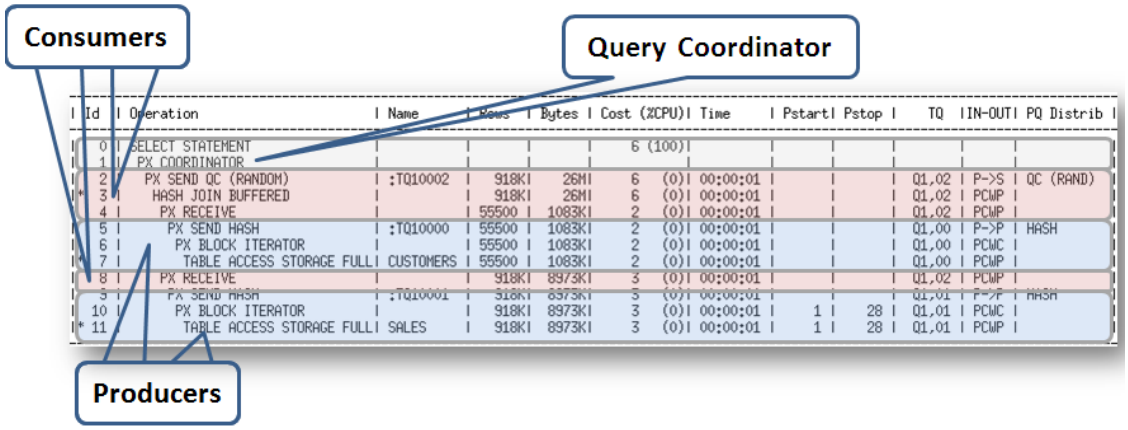
Figure31: Example plan output highlighting partition-based granules

Based on the SQL statement and the degree of parallelism, the Oracle Database decides whether block-based or partition-based granules lead to a more optimal execution; you cannot influence this behavior.

Producers and Consumers

In order to execute a statement in parallel efficiently parallel server processes actually work together in sets: one set is producing rows (producer) and one set is consuming the rows (consumer). For example in the plan in Figure32, the parallel join between the SALES and CUSTOMERS uses two sets of parallel server processes. The producers scan the two tables, applying any where clause predicates and send the resulting rows to the consumers (lines 9-11

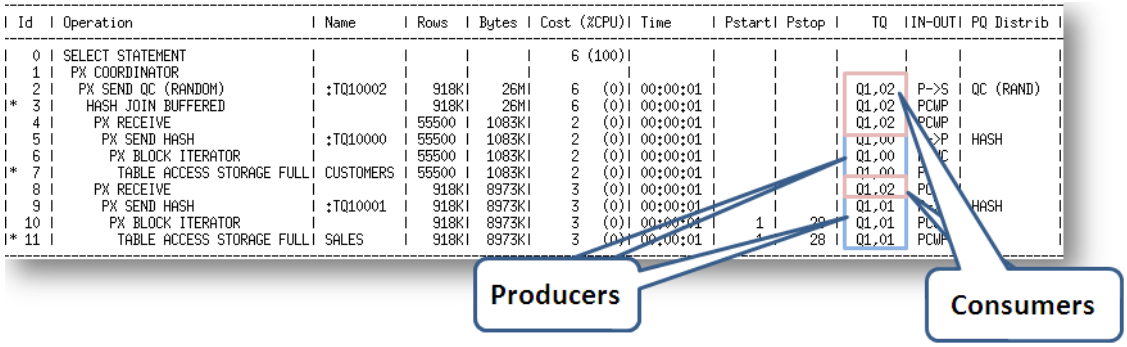
and lines 5-7). You can identify the producers because they do the operations below any PX SEND operation (line 9 & 5). The consumers complete the actual hash join and send the results to the QC (line 8 and lines 2-4). The consumers can be identified because they must do a PX RECEIVE before they can begin working (line 8 & 4) and they always finish by doing a PX SEND QC (line 2), which indicates they send the results to the QC.



Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time	Pstart	Pstop	TQ	IN-OUT	PQ Distrib
0	SELECT STATEMENT				6 (100)						
1	PX COORDINATOR										
2	PX SEND QC (RANDOM)	:TQ10002	918K	26M	6 (0)	00:00:01			Q1,02	P->S	QC (RAND)
* 3	HASH JOIN BUFFERED		918K	26M	6 (0)	00:00:01			Q1,02	PCWP	
4	PX RECEIVE		55500	1083K	2 (0)	00:00:01			Q1,02	PCWP	
5	PX SEND HASH	:TQ10000	55500	1083K	2 (0)	00:00:01			Q1,00	P->P	HASH
6	PX BLOCK ITERATOR		55500	1083K	2 (0)	00:00:01			Q1,00	PCWC	
7	TABLE ACCESS STORAGE FULL	CUSTOMERS	55500	1083K	2 (0)	00:00:01			Q1,00	PCWP	
8	PX RECEIVE		918K	8973K	3 (0)	00:00:01			Q1,02	PCWP	
9	PX SEND HASH	:TQ10001	918K	8973K	3 (0)	00:00:01			Q1,01	P->P	HASH
10	PX BLOCK ITERATOR		918K	8973K	3 (0)	00:00:01	1	28	Q1,01	PCWC	
* 11	TABLE ACCESS STORAGE FULL	SALES	918K	8973K	3 (0)	00:00:01	1	28	Q1,01	PCWP	

Figure32: Example plan output highlighting parallel producers and consumers

Similar information is shown in the TQ column. It shows which set of parallel server processes executed, which set of steps in the execution plan. In the plan above the Q1,00 set of parallel server processes (producers) scanned the `CUSTOMERS` table first. They then sent the resulting rows to the consumers (line 5) Q1,02. The Q1,00 set of parallel server processes then became the Q1,01 set of parallel server processes (which again are producers). The Q1,01 set scanned the `SALES` table and sent the resulting rows to the consumers(line9). The Q1,02 set of parallel server process (the consumers) accepted rows from the producers (line 8 & 4), completed the join (line 3) and sent the results to the query coordinator (2).



Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time	Pstart	Pstop	TQ	IN-OUT	PQ Distrib
0	SELECT STATEMENT				6 (100)						
1	PX COORDINATOR										
2	PX SEND QC (RANDOM)	:TQ10002	918K	26M	6 (0)	00:00:01			Q1,02	P->S	QC (RAND)
* 3	HASH JOIN BUFFERED		918K	26M	6 (0)	00:00:01			Q1,02	PCWP	
4	PX RECEIVE		55500	1083K	2 (0)	00:00:01			Q1,02	PCWP	
5	PX SEND HASH	:TQ10000	55500	1083K	2 (0)	00:00:01			Q1,00	P->P	HASH
6	PX BLOCK ITERATOR		55500	1083K	2 (0)	00:00:01			Q1,00	PCWC	
* 7	TABLE ACCESS STORAGE FULL	CUSTOMERS	55500	1083K	2 (0)	00:00:01			Q1,00	PCWP	
8	PX RECEIVE		918K	8973K	3 (0)	00:00:01			Q1,02	PCWP	
9	PX SEND HASH	:TQ10001	918K	8973K	3 (0)	00:00:01			Q1,01	P->P	HASH
10	PX BLOCK ITERATOR		918K	8973K	3 (0)	00:00:01	1	28	Q1,01	PCWC	
* 11	TABLE ACCESS STORAGE FULL	SALES	918K	8973K	3 (0)	00:00:01	1	28	Q1,01	PCWP	

Figure 33 Example plan output highlighting the TQ (table queue) column for producers and consumers

Data redistribution

In the example above two large tables `CUSTOMERS` and `SALES` are involved in the join. In order to process this join in parallel, a redistribution of rows becomes necessary between the producers and the consumers. The producers scan the tables based on block ranges and apply any where clause predicates and then send the resulting rows to the consumers, who will complete the join. The last two columns in the execution plan, `IN-OUT`, and `PQ Distrib` hold information about how the data is redistributed between the producers and consumers. The PQ Distrib column is the most useful column and has somewhat replaced the `IN-OUT` column.

The following five redistribution methods are the most common and you will see these names appearing in the PQ Distrib column of the execution plan.

HASH: Hash redistribution is very common in parallel execution in order to achieve an equal distribution among the parallel server processes. A hash function is applied to the join column and the result dictates which consumer parallel server process should receive the row.

BROADCAST: Broadcast redistribution happens when one of the two result sets in a join operation is much smaller than the other result set. Instead of redistributing rows from both result sets the database sends the smaller result set to all of the consumer parallel server processes in order to guarantee the individual servers are able to complete their join operation.

RANGE: Range redistribution is generally used for parallel sort operations. Individual parallel server processes work on data ranges so that the QC does not have to do any additional sorting but only present the individual parallel server processes results in the correct order.

KEY: Key redistribution ensures result sets for individual key values are clumped together. This is an optimization that is primarily used for partial partition-wise joins to ensure only one side in the join has to be redistributed.

ROUND ROBIN: Round-robin data redistribution can be the final redistribution operation before sending data to the requesting process. It can also be used in an early stage of a query when no redistribution constraints are required.

You may see a **LOCAL** suffix on the redistribution methods in a Real Application Clusters (RAC) database. LOCAL redistribution is an optimization in RAC to minimize interconnect traffic for inter-node parallel queries. In this case, the rows are distributed to only the consumers on the same RAC node.

In the plan in Figure 34 the producers send data to the consumers using a HASH redistribution method.

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time	Pstart	Pstop	TQ	IN-OUT	PQ Distrib
0	SELECT STATEMENT				6 (100)						
1	PX COORDINATOR										
2	PX SEND QC (RANDOM)	:TQ10002	918K	26M	6 (0)	00:00:01			Q1,02	P->S	QC (RAND)
*3	HASH JOIN BUFFERED		918K	26M	6 (0)	00:00:01			Q1,02	PCWP	
4	PX RECEIVE		55500	1083K	2 (0)	00:00:01			Q1,02	PCWP	
5	PX SEND HASH	:TQ10000	55500	1083K	2 (0)	00:00:01			Q1,00	P->P	HASH
6	PX BLOCK ITERATOR		55500	1083K	2 (0)	00:00:01			Q1,00	PCWC	
*7	TABLE ACCESS STORAGE FULL	CUSTOMERS	55500	1083K	2 (0)	00:00:01			Q1,00	PCWP	
8	PX RECEIVE		918K	8973K	3 (0)	00:00:01			Q1,02	PCWP	
9	PX SEND HASH	:TQ10001	918K	8973K	3 (0)	00:00:01			Q1,01	P->P	HASH
10	PX BLOCK ITERATOR		918K	8973K	3 (0)	00:00:01	1	28	Q1,01	PCWC	
*11	TABLE ACCESS STORAGE FULL	SALES	918K	8973K	3 (0)	00:00:01	1	28	Q1,01	PCWP	

Figure 34 Example plan output highlighting the row redistribution of parallel processing

You should also notice that on the lines in the plan where data redistribution takes place the value in the `IN-OUT` column is either `P->P` (lines 5 & 9) or `P->S` (line 2). `P->P` means that data is being sent from one parallel operation to another. `P->S` means that data is being sent from a parallel operation to serial operation. On line 2 the data is being sent to the QC, which is a single process, hence the `P->S`. However, if you see a `P->S` operation happening somewhere lower in the execution it may indicate you have a serialization point in the plan, which should be investigated. This could be caused by not having a parallel degree set on one or more of the objects accessed in the query.

Conclusion

The purpose of the Oracle Optimizer is to determine the most efficient execution plan for your queries. It makes these decisions based on the statistical information it has about your data and by leveraging Oracle database features such as hash joins, parallel query, and partitioning.

The explain plan is by far the most useful tool at our disposal when it comes to investigating why the Optimizer makes the decisions it makes. By breaking down the explain plan and reviewing the four key elements of: cardinality estimations, access methods, join methods, and join orders; you can determine if the execution plan is the best available plan.

Oracle Corporation, World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065, USA

Worldwide Inquiries
Phone: +1.650.506.7000
Fax: +1.650.506.7200

CONNECT WITH US

blogs.oracle.com/oracle

facebook.com/oracle

twitter.com/oracle

oracle.com

Hardware and Software, Engineered to Work Together