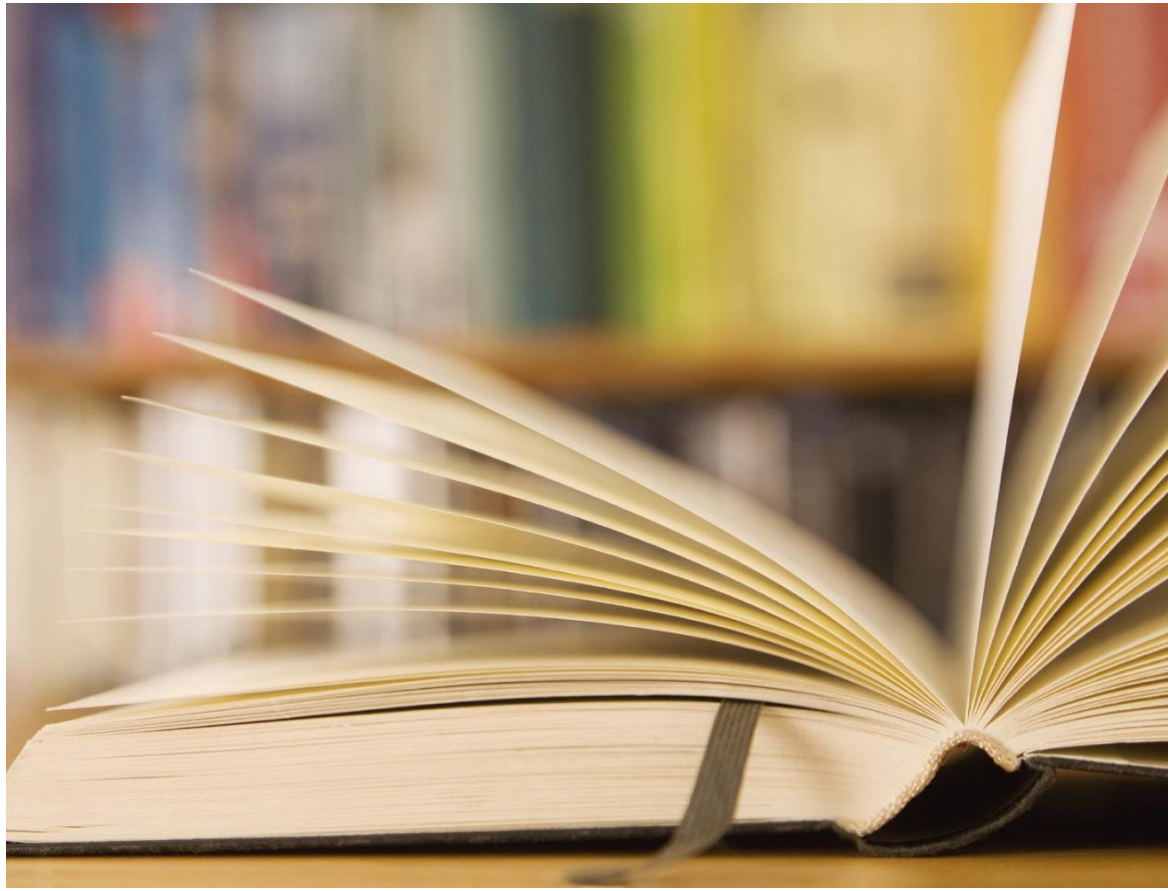
CSCI317 – Database Performance Tuning

Execution Plan

27 August 2020



Query Optimizer

Query Optimizer

- Query optimizer determines the most efficient **execution plan** for queries submitted to the server.
- It optimizes based on the statistical information it has about the data in the database and leveraging on the database features such as the various **join methods**, **access method**, parallel execution, partition pruning, etc. to generate the execution plan.
- Query optimizer does not guarantee to produce the best, optimal execution plan; several factors such as the skewing of data and complex expression may lead to inaccurate estimation.

Execution Plan

Execution Plan

- An execution plan contains detailed steps necessary to execute an SQL statement.
- The order of the operators and their implementations is decided by the query optimizer using a combination of **query transformations** and **physical optimization techniques**.
- An execution plan can be generated for a query using the command:

explain plan for …

Execution Plan - Example

An example to generate an execution plan for a query:

```
Explain plan for
Select   o.o_orderkye, o_o_totalprice, to_char(o.o_orderdate,'yyyy')
From     orders o
Where    o_o_totalprice = ( select  max(o_totalprice)
                            from    orders f
                            where to_char(f.o_orderdate,'yyyy') =
                                  to_char(o.o_orderdate,'yyyy') )
Order by to_char(o.o_orderdate,'yyyy');
```

```
SQL> spool ExampleA4T1Output.txt
SQL> @ExampleA4T1.sql
SQL> set linesize 130
SQL> set pagesize 0
SQL> --
SQL> -- Generate the execution plan for query 1
SQL> explain plan for
  2  select     o.o_orderkey, o.o_totalprice, to_char(o.o_orderdate,'yyyy')
  3  from       orders o
  4  where      o.o_totalprice = (
  5           select  max(o_totalprice)
  6           from    orders f
  7           where   to_char(f.o_orderdate,'yyyy') = to_char(o.o_orderdate,'yyyy')
  8           )
  9  order by to_char(o.o_orderdate,'yyyy');

Explained.
```

Execution Plan

- After the execution plan is generated, the plan can be obtained (displayed) using the DBMS_XPLAN package as follow:

```
Select      *
From        table(dbms_xplan.display);
```

- Note: The execution plan can be obtained using several PL/SQL interface. For more details on the package, please refer to “The Oracle Optimizer Explain the Explain Plan” at https://www.oracle.com/technetwork/database/bi-datawarehousing/twp-explain-the-explain-plan-052011-393674.pdf

Execution Plan

- An example to display the execution plan:

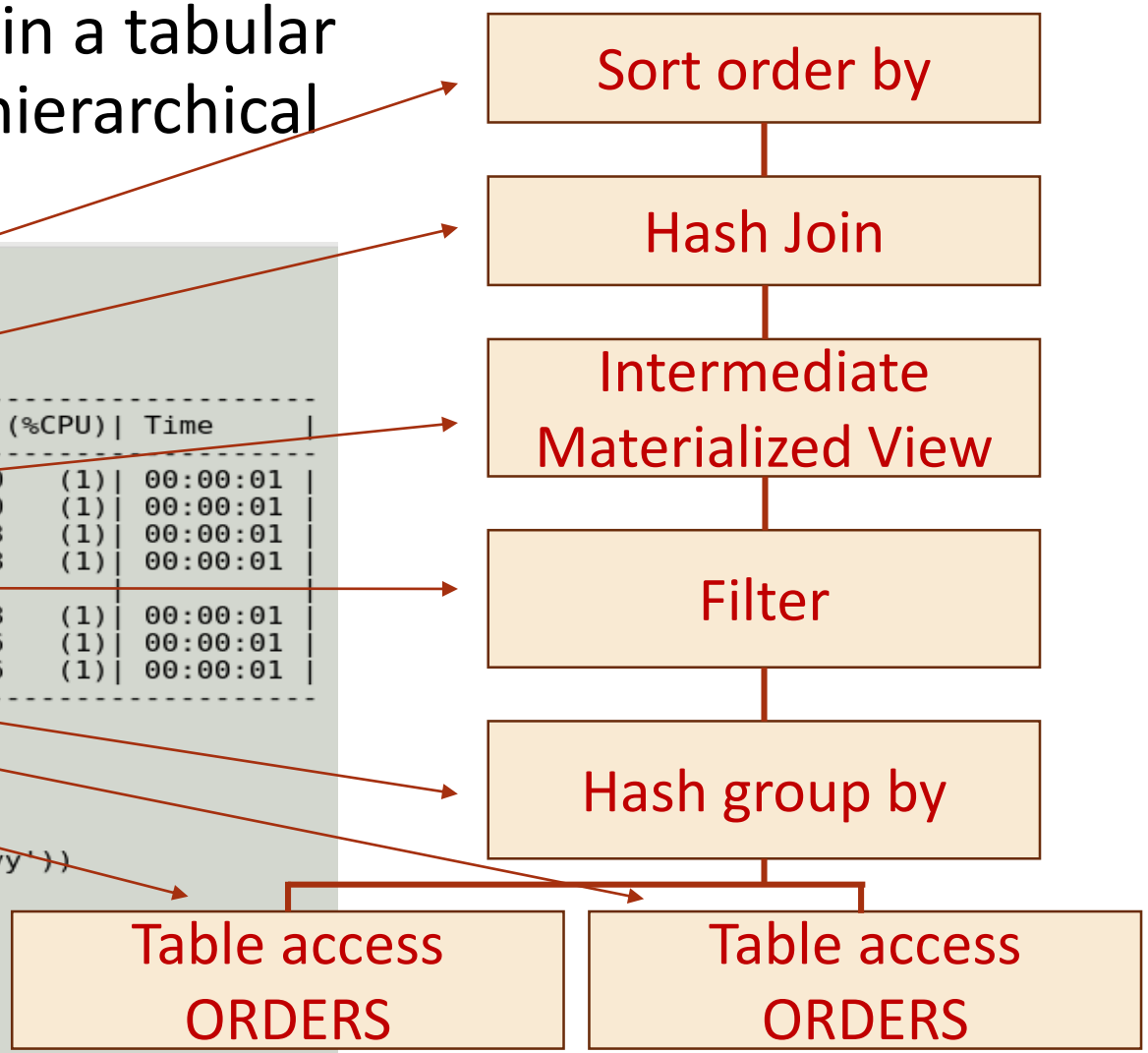
```
select    *
from      table(dbms_xplan.display);
```

```
SQL> -- Show the execution plan
SQL> select     *
  2  from       table(dbms_xplan.display);
Plan hash value: 777736422

-----------------------------------------------------------------------------------------------
| Id  | Operation               | Name    | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT        |         |  5001 |   253K|       |  7180   (1)| 00:00:01 |
|   1 |  SORT ORDER BY          |         |  5001 |   253K|   328K|  7180   (1)| 00:00:01 |
|*  2 |   HASH JOIN             |         |  5001 |   253K|    13M|  7113   (1)| 00:00:01 |
|   3 |    VIEW                 | VW_SQ_1 |   474K|  7873K|       |  2708   (1)| 00:00:01 |
|*  4 |     FILTER              |         |       |       |       |            |          |
|   5 |      HASH GROUP BY      |         |   474K|     9M|       |  2708   (1)| 00:00:01 |
|   6 |       TABLE ACCESS FULL | ORDERS  |   474K|     9M|       |  2696   (1)| 00:00:01 |
|   7 |    TABLE ACCESS FULL    | ORDERS  |   474K|    15M|       |  2696   (1)| 00:00:01 |
-----------------------------------------------------------------------------------------------

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

   2 - access("O"."O_TOTALPRICE"="MAX(O_TOTALPRICE)" AND
              "ITEM_1"=TO_CHAR(INTERNAL_FUNCTION("O"."O_ORDERDATE"),'yyyy'))
   4 - filter(MAX("O_TOTALPRICE")>=0)

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

25 rows selected.
```

Execution Plan

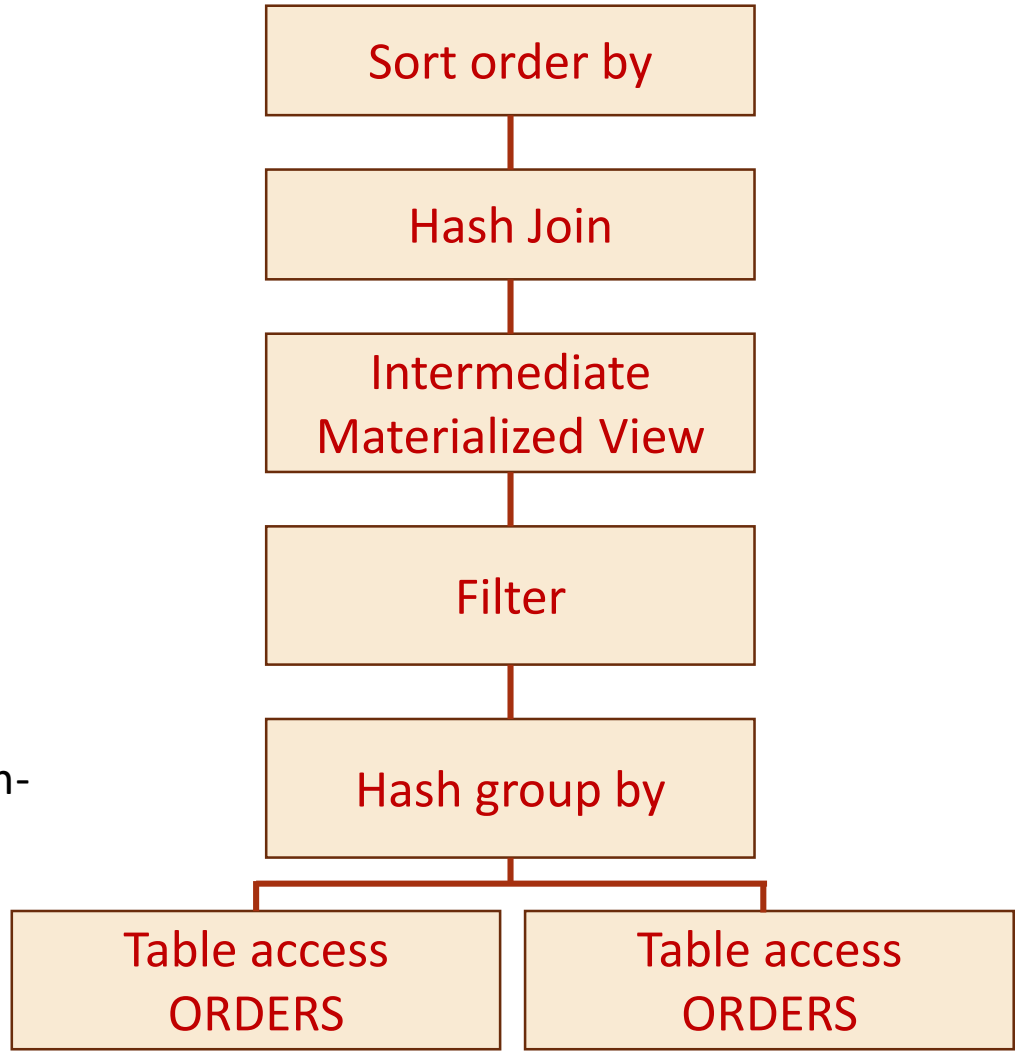
- The execution plan is commonly display in a tabular format, however, the actual plan is in a hierarchical structure, tree-shaped.

```
SQL> -- Show the execution plan
SQL> select     *
  2  from       table(dbms_xplan.display);
Plan hash value: 777736422

-----------------------------------------------------------------------------------------------
| Id  | Operation              | Name    | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT       |         |  5001 |   253K|       |  7180   (1)| 00:00:01 |
|   1 |  SORT ORDER BY         |         |  5001 |   253K|   328K|  7180   (1)| 00:00:01 |
|*  2 |   HASH JOIN            |         |  5001 |   253K|    13M|  7113   (1)| 00:00:01 |
|   3 |    VIEW                | VW_SQ_1 |   474K|  7873K|       |  2708   (1)| 00:00:01 |
|*  4 |     FILTER             |         |       |       |       |            |          |
|   5 |      HASH GROUP BY     |         |   474K|     9M|       |  2708   (1)| 00:00:01 |
|   6 |       TABLE ACCESS FULL| ORDERS  |   474K|     9M|       |  2696   (1)| 00:00:01 |
|   7 |    TABLE ACCESS FULL   | ORDERS  |   474K|    15M|       |  2696   (1)| 00:00:01 |
-----------------------------------------------------------------------------------------------

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

   2 - access("O"."O_TOTALPRICE"="MAX(O_TOTALPRICE)" AND
              "ITEM_1"=TO_CHAR(INTERNAL_FUNCTION("O"."O_ORDERDATE"),'yyyy'))
   4 - filter(MAX("O_TOTALPRICE")>=0)

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

25 rows selected.
```



Execution Plan

- The tabular representation of the execution plan is shown top-down, left-to-right order.
- When reading the execution tree, it should be read left-to-right, bottom-up.
- In the example shown, the plan is implemented by first:
  - i. Accessing the table ORDERS two times (one for the outer query and other for the sub-query);
  - ii. The rows produced by the table scans will then be hashed into temporary hash-table following the 'group by' condition;
  - iii. The temporary results are then filter to produce the rows that meet the condition specified into an intermediate materialized views;
  - iv. The results in the intermediate materialized views are then hash-join into final intermediate materialized view
  - v. The results in the final intermediate materialized view are then sorted according to the 'order by' attribute(s) and return to the user.



Execution Plan

Cost

Execution Plan - Cost

```
SQL> -- Show the execution plan
SQL> select     *
  2  from       table(dbms_xplan.display);
Plan hash value: 777736422

-----------------------------------------------------------------------------------------------
| Id  | Operation              | Name    | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT       |         |  5001 |   253K|       |  7180   (1)| 00:00:01 |
|   1 |  SORT ORDER BY         |         |  5001 |   253K|   328K|  7180   (1)| 00:00:01 |
|*  2 |   HASH JOIN            |         |  5001 |   253K|    13M|  7113   (1)| 00:00:01 |
|   3 |    VIEW                | VW_SQ_1 |   474K|  7873K|       |  2708   (1)| 00:00:01 |
|*  4 |     FILTER             |         |       |       |       |            |          |
|   5 |      HASH GROUP BY     |         |   474K|     9M|       |  2708   (1)| 00:00:01 |
|   6 |       TABLE ACCESS FULL| ORDERS  |   474K|     9M|       |  2696   (1)| 00:00:01 |
|   7 |    TABLE ACCESS FULL   | ORDERS  |   474K|    15M|       |  2696   (1)| 00:00:01 |
-----------------------------------------------------------------------------------------------

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

   2 - access("O"."O_TOTALPRICE"="MAX(O_TOTALPRICE)" AND
              "ITEM_1"=TO_CHAR(INTERNAL_FUNCTION("O"."O_ORDERDATE"),'yyyy'))
   4 - filter(MAX("O_TOTALPRICE")>=0)

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

25 rows selected.
```

The cost of the entire plan is indicated on line 0.

- The Oracle optimizer is a cost-based optimizer; it selects the execution plan with the lowest cost, where the cost represents the estimated resource usage for that plan.
- The optimizer's cost model accounts for the IO, CPU, and network resources that will be used by the query.
- The lower the cost, the better.

Execution Plan

Some Terminologies

Execution Plan

To understand the execution plan, it is important that we understand the following components (terminologies) in the plan:

- Cardinality
- Access method
- Join method
- Join type
- Join order

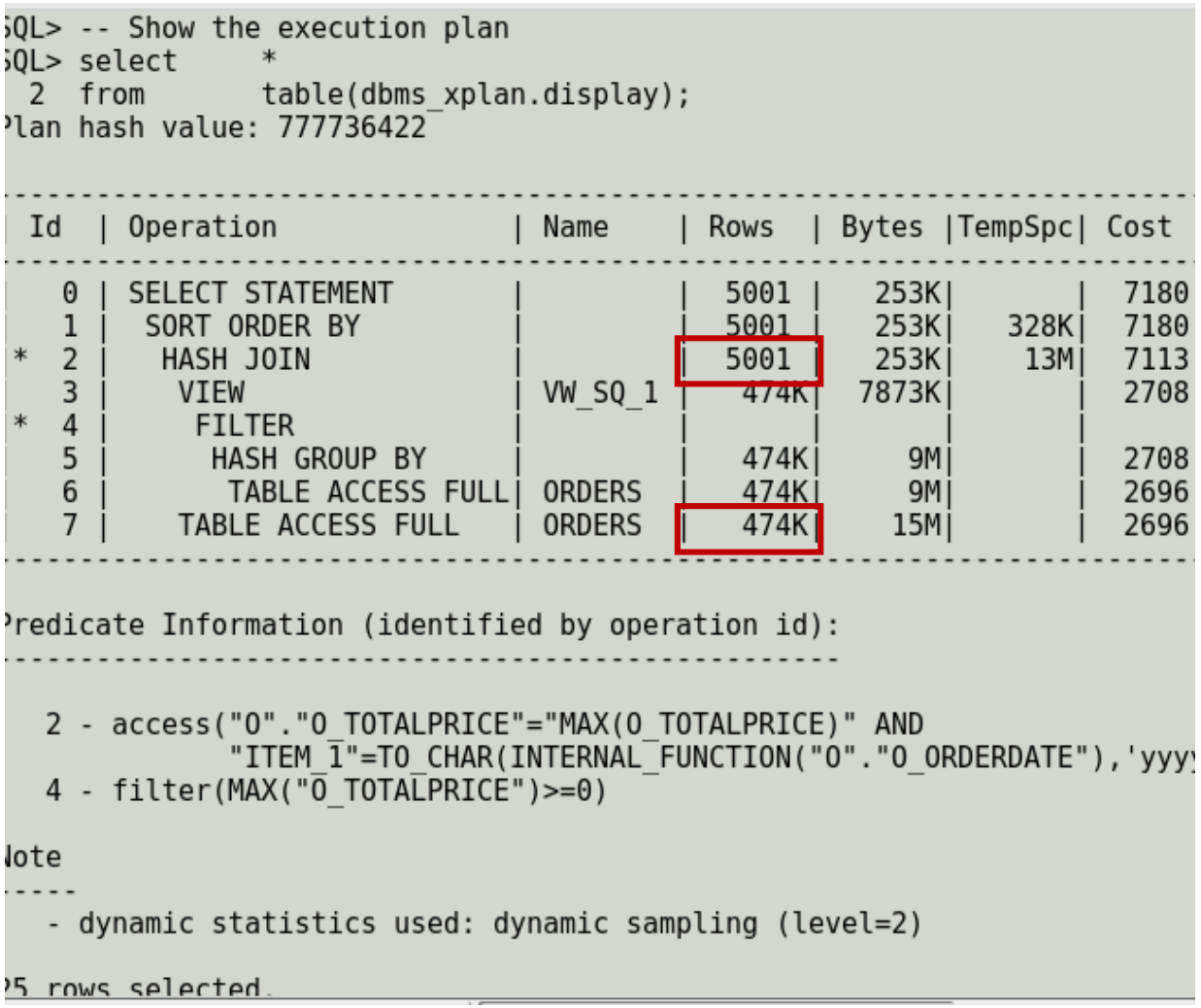
Execution Plan

- When there are multiple tables involved in the FROM clause of a query, the query optimizer must determine which join operation is most efficient for each pair.
- The optimizer must make decision based on the following interrelated operations – Cardinality, Access paths, Join methods, Join types, and join orders.

Execution Plan
Cardinality

Execution Plan - Cardinality

- **Cardinality** refers to the estimated number of rows coming out of each of the operations.
- The optimizer determines the cardinality for each operations based on a complex set of formulas that use as input both the statistics from the table as well as table columns.
- For example, one of the simplest formula is used when there is a single equality predicate (condition) in a single table query. Uniform distribution of the rows will be assumed, and the cardinality for the query is then obtained 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.



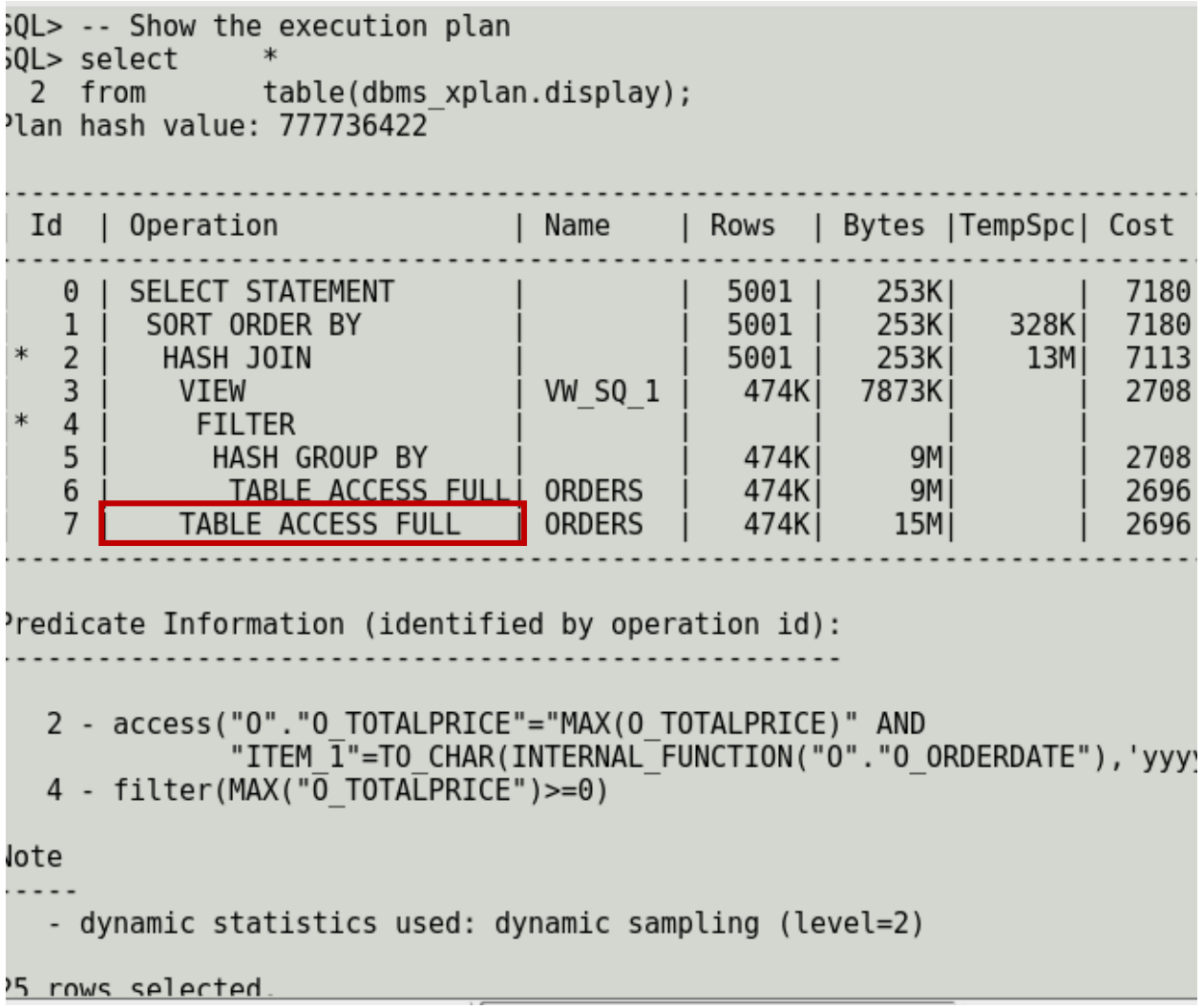
```
SQL> -- Show the execution plan
SQL> select     *
  2  from       table(dbms_xplan.display);
Plan hash value: 777736422

-------------------------------------------------------------------------------
| Id  | Operation              | Name    | Rows  | Bytes |TempSpc| Cost
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT       |         |  5001 |   253K|       |  7180
|   1 |  SORT ORDER BY         |         |  5001 |   253K|   328K|  7180
| *  2 |   HASH JOIN            |         |  5001 |   253K|    13M|  7113
|   3 |    VIEW                | VW_SQ_1 |   474K|  7873K|       |  2708
| *  4 |     FILTER             |         |       |       |       |
|   5 |      HASH GROUP BY     |         |   474K|     9M|       |  2708
|   6 |       TABLE ACCESS FULL| ORDERS  |   474K|     9M|       |  2696
|   7 |    TABLE ACCESS FULL   | ORDERS  |   474K|    15M|       |  2696
-------------------------------------------------------------------------------

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

   2 - access("O"."O_TOTALPRICE"="MAX(O_TOTALPRICE)" AND
              "ITEM_1"=TO_CHAR(INTERNAL_FUNCTION("O"."O_ORDERDATE"),'yyy
   4 - filter(MAX("O_TOTALPRICE")>=0)

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

25 rows selected.
```

Execution Plan

Access Method

Execution Plan – Access method

- The access method or access path shows how the data will be accessed from each table or index.
- Oracle supports nine common access methods:
  i. Full-table scan
  ii. Table accessed by ROWID
  iii. Index-unique scan
  iv. Index-range scan and Index-range scan descending
  v. Index-skip scan
  vi. Full-index scan
  vii. Fast-full-index scan
  viii. index join
  ix. Bitmap index



```
SQL> -- Show the execution plan
SQL> select     *
  2  from       table(dbms_xplan.display);
Plan hash value: 777736422

-------------------------------------------------------------------------------
| Id  | Operation              | Name    | Rows  | Bytes |TempSpc| Cost
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT       |         |  5001 |   253K|       |  7180
|   1 |  SORT ORDER BY         |         |  5001 |   253K|   328K|  7180
| *  2 |   HASH JOIN            |         |  5001 |   253K|    13M|  7113
|   3 |    VIEW                | VW_SQ_1 |   474K|  7873K|       |  2708
| *  4 |     FILTER             |         |       |       |       |
|   5 |      HASH GROUP BY     |         |   474K|     9M|       |  2708
|   6 |       TABLE ACCESS FULL| ORDERS  |   474K|     9M|       |  2696
|   7 |    TABLE ACCESS FULL   | ORDERS  |   474K|    15M|       |  2696
-------------------------------------------------------------------------------

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

   2 - access("O"."O_TOTALPRICE"="MAX(O_TOTALPRICE)" AND
              "ITEM_1"=TO_CHAR(INTERNAL_FUNCTION("O"."O_ORDERDATE"),'yyy
   4 - filter(MAX("O_TOTALPRICE")>=0)

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

25 rows selected.
```

Execution Plan – Access method

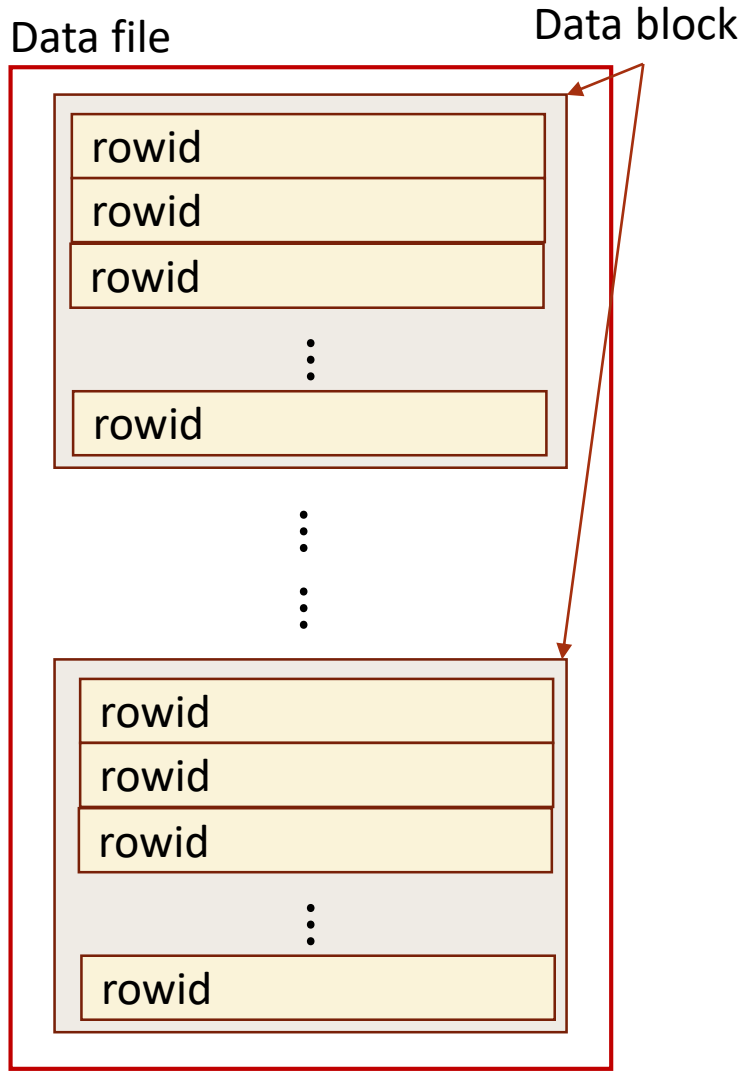
Full-table scan

- Reads all rows from a table and filters out those that do not meet the where clause predicates.
- A full-table scan is selected if
  - a large portion of the rows in the table must be accessed or
  - no indexes exist or
  - the index cannot be used or
  - if the cost is the lowest.

Execution Plan – Access method

Table accessed by ROWID

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



Execution Plan – Access method

Index-unique scan

- Index-unique scan returns only **one** row that matches the equality predicate on a unique B*-tree index or an index created as a result of a primary key constraint.
- Index-unique scan will be used when there is an equality predicate on a unique B*-tree index or a primary key index.

Execution Plan – Access method

Index-range scan or Index-range scan descending

- 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.
- Data is returned in the ascending order of the index columns.
- Index-range scan descending is conceptually the same as index-range scan except that data is returned in the descending order of the index columns.
- Index-range scan descending is used when an ‘ORDER BY …. DESCENDING’ clause can be satisfied by an index.

Execution Plan – Access method

Index skip scan

- In order for an index to be used, the prefix of the index key 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.

Execution Plan – Access method

Full-index scan

- Contrary to its name, a full-index scan does not read every block in the index structure.
- An index full scan processes all of the leaf blocks of an index, but only enough of the branch books to find the first leaf block.
- Full-index scan is used when all of the columns necessary to satisfy the statement are in the index and it is cheaper than scanning the table.

Execution Plan – Access method

Full-index scan (…continue…)

- Full-index scan may be used in any of the following situations:
  i. 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).
  ii. The query requires a sort merge join and all of the columns references in the query are in the index.
  iii. Order of the columns referenced in the query matches the order of the leading index columns.
  iv. A GROUP BY clause is present in the query, and the columns in the GROUP BY clause are present in the index.

Execution Plan – Access method

Full-index scan (…continue…)

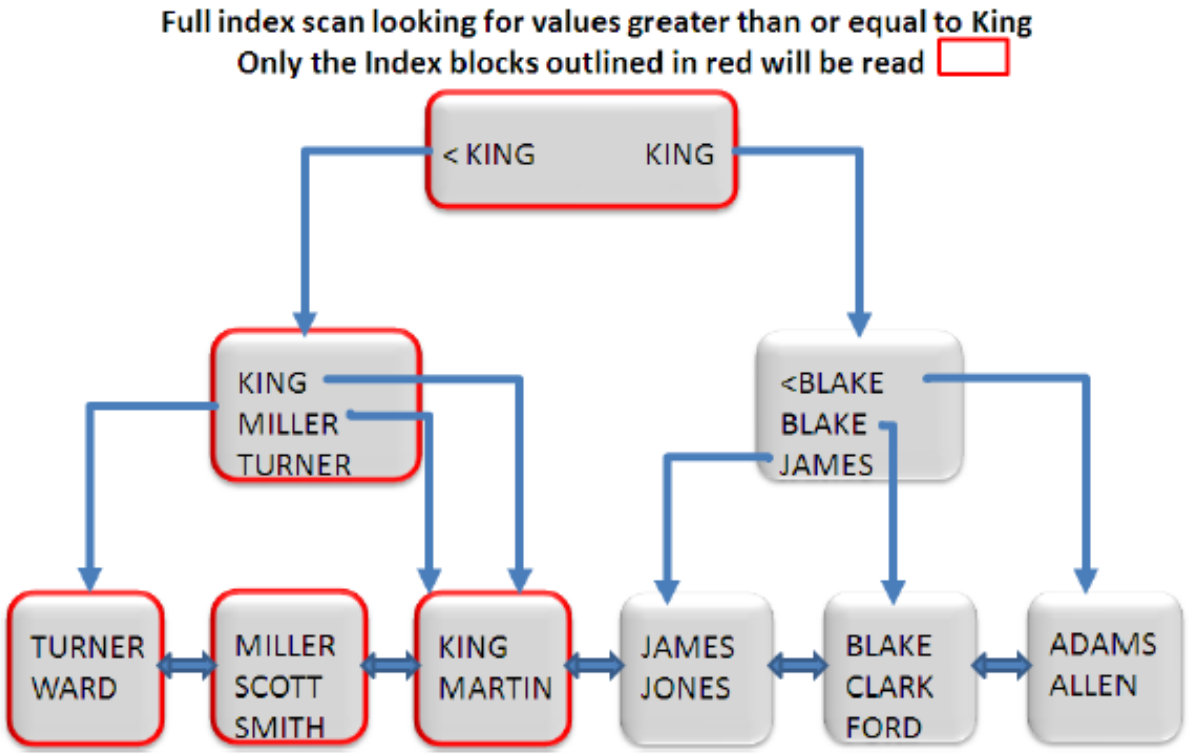


Figure 16: Processing of an INDEX FULL SCAN

https://www.oracle.com/technetwork/database/bi-datawarehousing/twp-explain-the-explain-plan-052011-393674.pdf

Execution Plan – Access method

Fast-full-index scan

- Fast-full-index scan is an alternative to 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.
- Unlike a full-index scan, fast-full-index scan will read all of the blocks in the index using multiblock reads.

Execution Plan – Access method

Index join

- It 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.

Execution Plan – Access method

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.


Execution Plan
Join Method

Execution Plan – Join Method

- Oracle support the following join methods:
  i. Hash joins
  ii. Nested-Loops joins
  iii. Sort-merge joins
  iv. Cartesian join

Execution Plan – Join Method

Hash join

- 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.
- The optimizer then scans the large table, and performs the same hashing algorithm on the join column(s).
- The optimizer then probes the previously built hash table for each value and if they match, it returns a row.
- Hash joins require an equijoin predicate, a predicate comparing values from one table with values from the other table using the equals operator '='.

Execution Plan – Join Method

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 such as table look up.
- For every row in the first table (the outer table), Oracle accesses all the rows in the second table (the inner table).

Execution Plan – Join Method

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 sort-merge join consists of two steps:
  i. Sort join operation: both the inputs are sorted on the join key
  ii. Merge join operation: the sorted lists are merged together.

Execution Plan – Join Method

- 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.

Execution Plan – Join Method

Cartesian joins

- 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.
- Cartesian join is 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.

Execution Plan
Join Type

Execution Plan – Join type

Oracle offers several join types:

i. Inner join (or simply join)
ii. Outer join
iii. Full outer join
iv. Anti join
v. Semi join

Execution Plan – Join type

Inner Join (or simply join)

- Oracle inner join combines the output from exactly two row sources, such as tables or views, and returns one row source.
- The returned row source is the data set.
- Inner joins are either **equijoins** or **nonequijoins**.
- An equijoin is an inner join whose **join condition** contains an equality operator.
- A nonequijoin is an inner join whose **join condition** contains an operator that is not an equality operator.

Execution Plan – Join type

- A **join condition** compares two row sources using an expression. The join condition defines the relationship between the tables. If the statement does not specify a join condition, then the optimizer performs a Cartesian join, matching every row in one table with every row in the other table.
- Inner join is used when multiple tables in the WHERE (non-ANSI) or FROM … JOIN (ANSI) clause of a SQL statement with join condition specified.

Execution Plan – Join type

Outer Join

- 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, the predicate T1.x = T2.x(+) indicates T1 is the left table whose non-joining rows will be retained.
- In 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).

Execution Plan – Join type

Anti join

- An Anti-join between two tables returns rows from the first table where no matches are found in the second table.
- An anti-join is essentially the opposite of a semi-join: While a semi-join returns on copy of each row in the first table for which at least one match is found, an anti-join returns one copy of each row in the first table for which no match is found.
- Anti-joins are written using the NOT EXISTS or NOT IN constructs. These two constructs differ in how they handle nulls - a subtle but very important distinction.

Execution Plan – Join type

Semi join

- A semi-join between two tables return rows from the first table where one or more matches are found in the second table.
- The difference between a semi-join and a conventional join is that rows in the first table will be returned at most once. Even if the second table contains two matches for a row in the first table, only one copy of the row will be returned. Semi-joins are written using the EXITS or IN constructs

Execution Plan – Join type

- **Note:** There are a few situations in which the optimizer cannot use a semi-join access path. If the query contains a DISTINCT operation (explicitly or implicitly through a set operator such as UNION) then the optimizer cannot transform EXISTS or IN clauses into something that could use a semi-join access path. The same is true if the EXISTS or IN clause is part of an OR branch.

Execution Plan – Join type

- If a semi-join access path is not possible because of a DISTINCT operation or an OR predicate, or if instance parameters specify a semi-join should not be used, then the optimizer will choose an alternate data access path. This usually amounts to performing a conventional join and sorting out the duplicates, or scanning the first table and using the second table as filter.


Execution Plan
Join Order

Execution Plan – 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, we look at the indentation of the tables in the operation column.
- In the example shown on the right, the first table access of the table ORDERS (line 7) is for the outer query, and the second table access of the table ORDERS (line 6, indented) is for the inner query (sub-query).

```
SQL> -- Show the execution plan
SQL> select     *
  2  from       table(dbms_xplan.display);
Plan hash value: 777736422

-------------------------------------------------------------------------------
| Id  | Operation              | Name    | Rows  | Bytes |TempSpc| Cost
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT       |         |  5001 |   253K|       |  7180
|   1 |  SORT ORDER BY         |         |  5001 |   253K|   328K|  7180
| *  2 |   HASH JOIN            |         |  5001 |   253K|    13M|  7113
|   3 |    VIEW                | VW_SQ_1 |   474K|  7873K|       |  2708
| *  4 |     FILTER             |         |       |       |       |
|   5 |      HASH GROUP BY     |         |   474K|     9M|       |  2708
|   6 |       TABLE ACCESS FULL| ORDERS  |   474K|     9M|       |  2696
|   7 |    TABLE ACCESS FULL   | ORDERS  |   474K|    15M|       |  2696
-------------------------------------------------------------------------------

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

   2 - access("O"."O_TOTALPRICE"="MAX(O_TOTALPRICE)" AND
              "ITEM_1"=TO_CHAR(INTERNAL_FUNCTION("O"."O_ORDERDATE"),'yyyy
   4 - filter(MAX("O_TOTALPRICE")>=0)

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

25 rows selected.
```

Execution Plan – Join Order

- In the second example shown on the right, the ORDERS and CUSTOMER tables will be joined first.
- The result (merge join cartesian) is then hash join to the third table LINEITEM.

```
SQL> l
  1  explain plan for
  2  select c_name, o_orderkey, l_suppkey
  3  from customer, orders, lineitem
  4  where o_custkey = o_custkey
  5* and o_orderkey = l_orderkey
SQL> /
```

```
SQL> select * from table(dbms_xplan.display);
Plan hash value: 350187101

-------------------------------------------------------------------------------------------------
| Id  | Operation             | Name     | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |          |    79G|  2977G|       |   161M  (1)| 01:45:17 |
|*  1 |  HASH JOIN            |          |    79G|  2977G|    37M|   161M  (1)| 01:45:17 |
|   2 |   TABLE ACCESS FULL   | LINEITEM |  1800K|    17M|       | 12145   (1)| 00:00:01 |
|   3 |   MERGE JOIN CARTESIAN|          |    20G|   565G|       |   121M  (1)| 01:18:56 |
|   4 |    TABLE ACCESS FULL  | CUSTOMER | 45000 |   834K|       |   389   (0)| 00:00:01 |
|   5 |    BUFFER SORT        |          |   450K|  4833K|       |   121M  (1)| 01:18:56 |
|   6 |     TABLE ACCESS FULL | ORDERS   |   450K|  4833K|       |  2694   (1)| 00:00:01 |
-------------------------------------------------------------------------------------------------

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

   1 - access("O_ORDERKEY"="L_ORDERKEY")

18 rows selected.
```

Execution Plan – Join Order

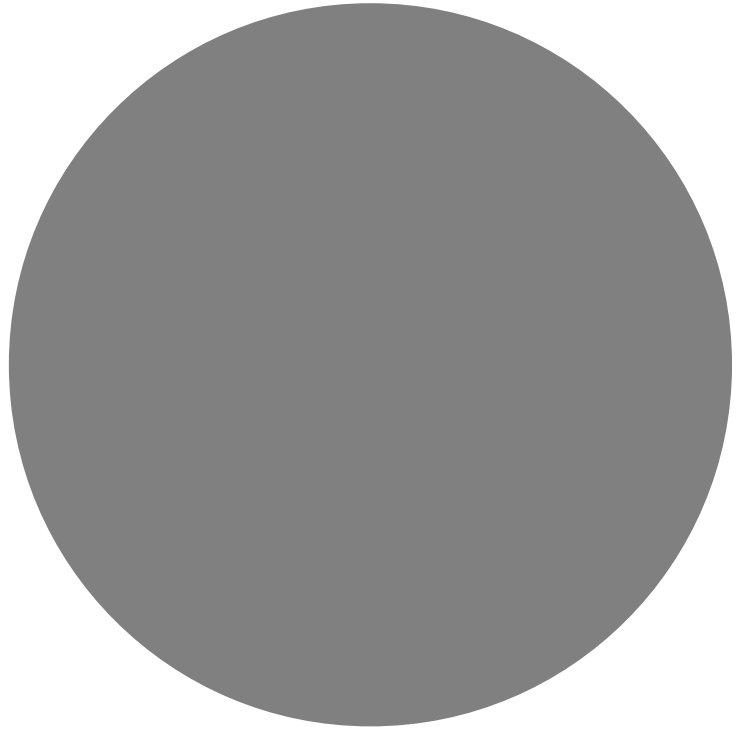
- The join order is determined based on cost, which is strongly influenced by the cardinality estimates and the access paths available.
- The Optimizer will also always adhere to some basic rules:
  i. Joins that result in at most one row always go first. The Optimizer can determine this based on UNIQUE and PRIMARY KEY constraints on the tables.
  ii. 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 do not satisfy the join condition can be added to the result set correctly.

Execution Plan

- When a subquery has been converted into an antijoin or semijoin, the table 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.

Example

Analyzing the Execution Plan



Example – Analysing Execution Plan

We have the following two different construct of queries that finds all top paid orders in each year and displays order key, total prices, and year of all top paid orders in each year:

```
select  o.o_orderkey, o.o_totalprice, to_char(o.o_orderdate,'yyyy')
from    orders o
where   o.o_totalprice = (
        select  max(o_totalprice)
        from    orders f
        where   to_char(f.o_orderdate,'yyyy') = to_char(o.o_orderdate,'yyyy')
        )
order by to_char(o.o_orderdate,'yyyy');
```

Example – Analysing Execution Plan

```
select  o.o_orderkey, o.o_totalprice, to_char(o.o_orderdate,'yyyy')
from    orders o
where   0 = (
        select  max(f.o_totalprice)
        from    orders f
        where   to_char(f.o_orderdate,'yyyy') = to_char(o.o_orderdate, 'yyyy')
        ) - o.o_totalprice
order by to_char(o.o_orderdate,'yyyy');
```

Question: Which of the two queries is more efficient?

Example – Analysing Execution Plan

- To answer or find out the decision, we will make use of the Optimizer execution plan.
- First, we generate the execution plan for each of the queries and analyse the cost determined by the optimizer.

Example – Analysing Execution Plan

```
SQL> @ExampleA4T1.sql
SQL> set linesize 130
SQL> set pagesize 0
SQL> --
SQL> -- Generate the execution plan for query 1
SQL> explain plan for
  2  select  o.o_orderkey, o.o_totalprice, to_char(o.o_orderdate,'yyyy')
  3  from    orders o
  4  where   o.o_totalprice = (
  5          select  max(o_totalprice)
  6          from    orders f
  7          where   to_char(f.o_orderdate,'yyyy') = to_char(o.o_orderdate,'yyyy')
  8          )
  9  order by to_char(o.o_orderdate,'yyyy');

Explained.
```

```
SQL> -- Show the execution plan
SQL> select  *
  2  from    table(dbms_xplan.display);
Plan hash value: 777736422

-------------------------------------------------------------------------------------------------
| Id  | Operation              | Name    | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT       |         |  5001 |   253K|       |  7180   (1)| 00:00:01 |
|   1 |  SORT ORDER BY         |         |  5001 |   253K|   328K|  7180   (1)| 00:00:01 |
|*  2 |   HASH JOIN            |         |  5001 |   253K|    13M|  7113   (1)| 00:00:01 |
|   3 |    VIEW                | VW_SQ_1 |   474K|  7873K|       |  2708   (1)| 00:00:01 |
|*  4 |     FILTER             |         |       |       |       |            |          |
|   5 |      HASH GROUP BY     |         |   474K|     9M|       |  2708   (1)| 00:00:01 |
|   6 |       TABLE ACCESS FULL| ORDERS  |   474K|     9M|       |  2696   (1)| 00:00:01 |
|   7 |    TABLE ACCESS FULL   | ORDERS  |   474K|    15M|       |  2696   (1)| 00:00:01 |
-------------------------------------------------------------------------------------------------

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

   2 - access("O"."O_TOTALPRICE"="MAX(O_TOTALPRICE)" AND
              "ITEM_1"=TO_CHAR(INTERNAL_FUNCTION("O"."O_ORDERDATE"),'yyyy'))
   4 - filter(MAX("O_TOTALPRICE")>=0)

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

25 rows selected.
```

Example – Analysing Execution Plan

```
SQL> -- Generate the execution plan for query 2
SQL> explain plan for
  2  select  o.o_orderkey, o.o_totalprice, to_char(o.o_orderdate,'yyyy')
  3  from    orders o
  4  where   0 = (
  5          select  max(f.o_totalprice)
  6          from    orders f
  7          where   to_char(f.o_orderdate,'yyyy') = to_char(o.o_orderdate, 'yyyy')
  8          ) - o.o_totalprice
  9  order by to_char(o.o_orderdate,'yyyy');

Explained.
```

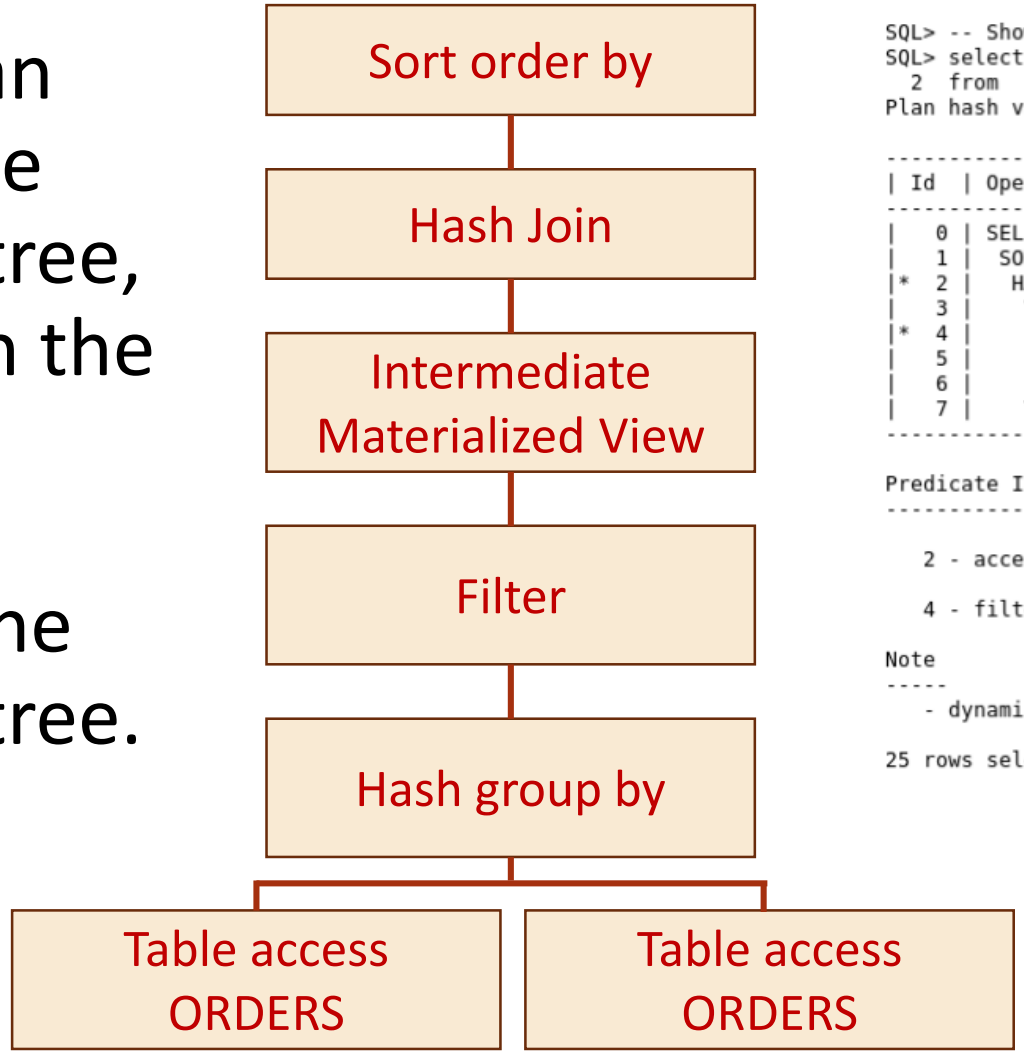
```
SQL> -- Show the execution plan
SQL> select  *
  2  from    table(dbms_xplan.display);
Plan hash value: 1152467146

-----------------------------------------------------------------------------------------
| Id  | Operation             | Name    | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |         |    22M|  1115M|       |   302K  (3)| 00:00:12 |
|   1 |  SORT ORDER BY        |         |    22M|  1115M|  1383M|   302K  (3)| 00:00:12 |
|*  2 |   HASH JOIN           |         |    22M|  1115M|    13M| 13085  (46)| 00:00:01 |
|   3 |    VIEW               | VW_SQ_1 |   474K|  7873K|       |  2708   (1)| 00:00:01 |
|   4 |     HASH GROUP BY     |         |   474K|     9M|       |  2708   (1)| 00:00:01 |
|   5 |      TABLE ACCESS FULL| ORDERS  |   474K|     9M|       |  2696   (1)| 00:00:01 |
|   6 |    TABLE ACCESS FULL  | ORDERS  |   474K|    15M|       |  2696   (1)| 00:00:01 |
-----------------------------------------------------------------------------------------

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

   2 - access("ITEM_0"=TO_CHAR(INTERNAL_FUNCTION("O"."O_ORDERDATE"),'yyyy'))
       filter("MAX(F.O_TOTALPRICE)"-"O"."O_TOTALPRICE"=0)

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

23 rows selected.
```

Example – Analysing Execution Plan

- Of course by looking at the final cost (line 0) of the two execution plan, we know the implementation of the first query is more efficient ☺
- However, for your assignment, this is not sufficient. You need to explain or justify your answer.
- One way to achieve that is to explain the execution plan!

Example – Analysing Execution Plan

- First, we can produce the execution tree, and explain the execution operation based on the execution tree.




```
SQL> -- Show the execution plan
SQL> select  *
  2  from    table(dbms_xplan.display);
Plan hash value: 777736422

-------------------------------------------------------------------------------------------------
| Id  | Operation              | Name    | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT       |         |  5001 |   253K|       |  7180   (1)| 00:00:01 |
|   1 |  SORT ORDER BY         |         |  5001 |   253K|   328K|  7180   (1)| 00:00:01 |
|*  2 |   HASH JOIN            |         |  5001 |   253K|    13M|  7113   (1)| 00:00:01 |
|   3 |    VIEW                | VW_SQ_1 |   474K|  7873K|       |  2708   (1)| 00:00:01 |
|*  4 |     FILTER             |         |       |       |       |            |          |
|   5 |      HASH GROUP BY     |         |   474K|     9M|       |  2708   (1)| 00:00:01 |
|   6 |       TABLE ACCESS FULL| ORDERS  |   474K|     9M|       |  2696   (1)| 00:00:01 |
|   7 |    TABLE ACCESS FULL   | ORDERS  |   474K|    15M|       |  2696   (1)| 00:00:01 |
-------------------------------------------------------------------------------------------------

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

   2 - access("O"."O_TOTALPRICE"="MAX(O_TOTALPRICE)" AND
              "ITEM_1"=TO_CHAR(INTERNAL_FUNCTION("O"."O_ORDERDATE"),'yyyy'))
   4 - filter(MAX("O_TOTALPRICE")>=0)

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

25 rows selected.
```

Example – Analysing Execution Plan

- From the execution tree shown above, the plan is implemented by first:
  i. Accessing the table ORDERS two times (one for the outer query and other for the sub-query). These operations produce a cost of 2696 units.
  ii. The rows produced by the table scans will then be hashed into temporary hash-table following the ‘group by’ condition. This operation increase the cost to 2708 units.
  iii. The temporary results are then filter to produce the rows that meet the condition specified into an intermediate materialized views. The cost of operation of filtering the result into intermediate materialized views is part of the 2708 units.

Example – Analysing Execution Plan

iv. The results in the intermediate materialized views are then hash-join into final intermediate materialized view. The cost of the hash join operation is substantial. The total cost of the operation so far is now increase to 7113 units.

v. The results in the final intermediate materialized view are then sorted according to the ‘order by’ attribute(s) and return to the user. This final sorting (sort order by) operation add additional of 67 units to the total cost.

vi. The final total cost for this implementation is 7180 units.

Reference

- https://www.oracle.com/technetwork/database/bi-datawarehousing/twp-explain-the-explain-plan-052011-393674.pdf
- https://docs.oracle.com/database/121/TGSQL/tgsql_join.htm#TGSQL94680