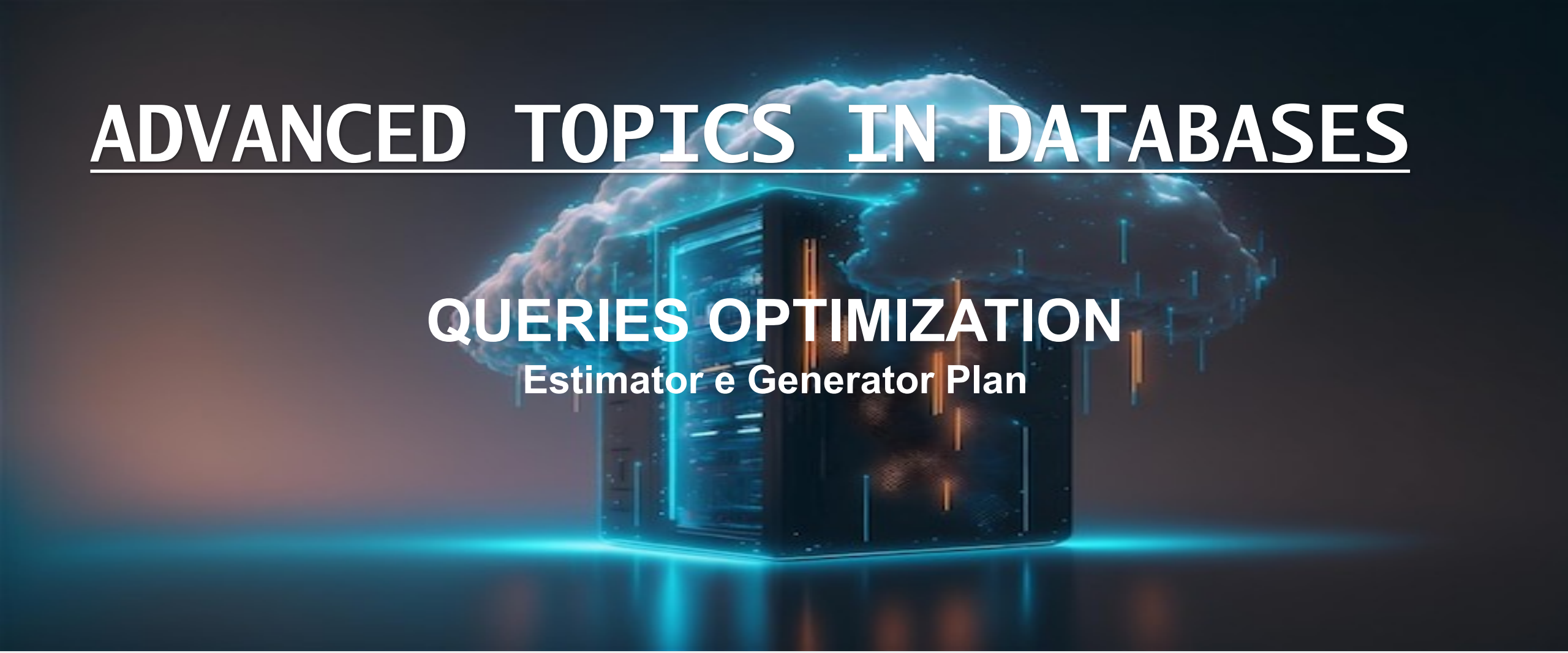


ADVANCED TOPICS IN DATABASES

QUERIES OPTIMIZATION

Estimator e Generator Plan

Master in Informatics Engineering
Data Engineering

Informatics Engineering Department

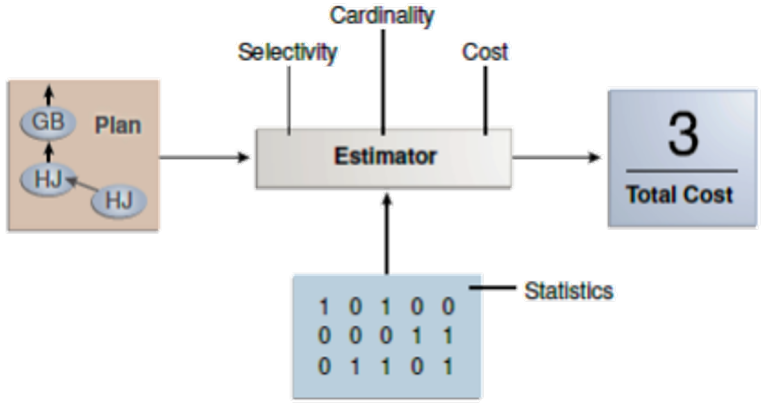
ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Query Optimizer

Estimator

Estimator

- The estimator is the component of the optimizer that determines the overall cost of a given execution plan.
- The estimator uses three different measures to determine cost:
  - Selectivity - The percentage of rows in the row set that the query selects, with 0 meaning no rows and 1 meaning all rows.
    - Selectivity is tied to a query predicate, such as WHERE last_name LIKE 'A%', or a combination of predicates.



Estimator

- Cardinality - is the number of rows returned by each operation in an execution plan.
  - The optimizer determines the cardinality for each operation based on a complex set of formulas that use both table and column level statistics, or dynamic statistics, as input.
  - The optimizer uses one of the simplest formulas when a single equality predicate appears in a single-table query, with no histogram.
    - 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.**

```
SELECT first_name, last_name
FROM employees
WHERE salary='10200';
```

- employees table contains 107 rows.
- The current database statistics indicate that the number of distinct values in the salary column is 58.

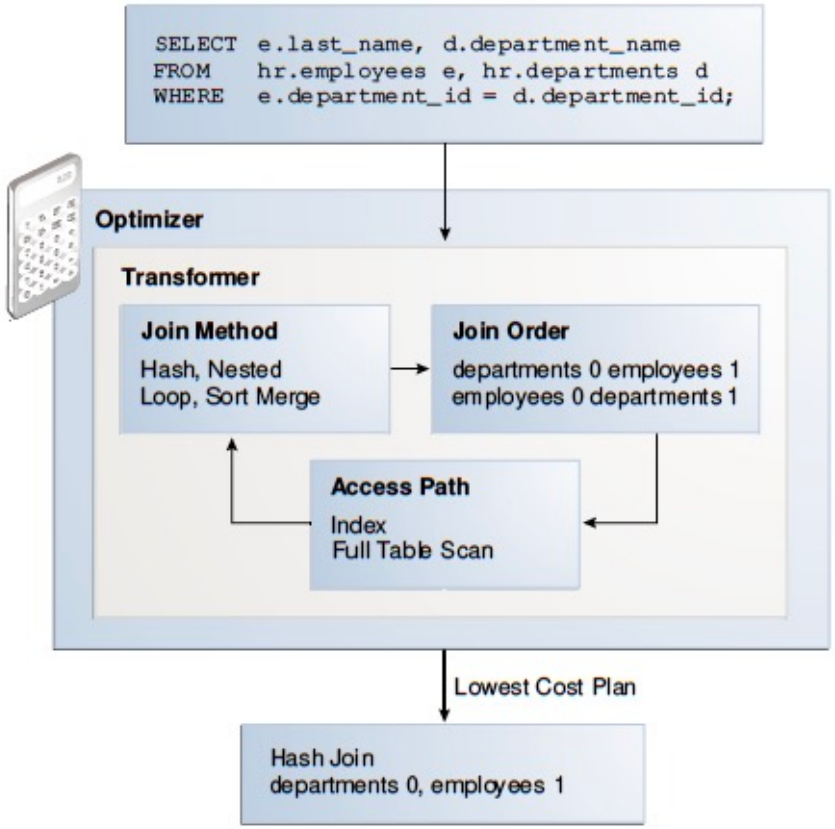
Cardinality = 2 (107/58=1.84).

Estimator

- Cost -This measure represents units of work or resource used.
  - The query optimizer uses :
    - System resources: disk I/O, CPU usage, and memory usage as units of work.
    - Estimated number of rows returned (cardinality).
    - Size of the initial of the data
    - Access structures

Plan Generator

- The plan generator explores various plans for a query block by trying out different **access paths, join methods, and join orders.** The goal is to choose the plan with lowest cost



Plan Generator

Access paths – are ways in which data is retrieved from the database.

Join methods - To join each pair of row sources.

- Oracle Database must perform a join operation.
- Join methods include nested loop, sort merge and hash join.
- Oracle joins two of tables and then joins the resulting row source to the nest table. This process continues all tables are joined into the result

Plan Generator –Access Paths

- The access paths that Oracle can use to locate and retrieval any row in table are:
  - **Full Table Scan**- is one of the access methods that are used by Optimizer. All the blocks in the table are scanned and ‘where’ filter conditions are applied and the lines providing the filter condition are returned.
  - **Rowid Scan-** RowID is the physical place of a line. It’s the quickest way to access data.
  - **Index Scan** - Index range scan is one of the most common accessing methods.
    - Oracle accesses the index records, finds the rowid equivalents and reaches the table

Access Paths – Full Table Scan

When the optimizer considers a full table scan

- **No index exists**
- **Small number of rows**
  - The cost of full table scan is less than index range scan due to small table.
- **When query processed SELECT COUNT(*), nulls existed in the column**
- **The query is unselective**
  - The number of return rows is too large and takes nearly 100% in the whole table. These rows are unselective.
- **The table statistics does not update**
  - The number of rows in the table is higher than before, but table statistics haven't been updated yet.
  - The optimizer can't correctly estimate that using the index is faster.
- **A full table scan hint**
  - The hint lets optimizer to use full table scan.

```
SELECT /*+ FULL (hr_emp) */ last_name
FROM employees hr_emp;
```

Access Paths – Rowid Scan

- A rowid is an internal representation of the storage location of data.
- The rowid of a row specifies the data file and data block containing the row and the location of the row in that block.
- To acess a table by rowid, oracle database first obtains the rowids of the selected rows, either from the statement WHERE clause or through an index scan;

  After, Oracle Database locates each selected row in the table based on its rowid

```
SELECT *
FROM   employees
WHERE  employee_id > 190;
```

```
-------------------------------------------------------------------------------
| 0| SELECT STATEMENT                    |             |  |    |2(100)|        |
| 1|  TABLE ACCESS BY INDEX ROWID BATCHED|EMPLOYEES    |16|1104|2  (0)|00:00:01|
|*2|   INDEX RANGE SCAN                  |EMP_EMP_ID_PK|16|    |1  (0)|00:00:01|
-------------------------------------------------------------------------------

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

   2 - access("EMPLOYEE_ID">190)
```

Access Paths – Index Scan

- In index scan, data is retrieved by traversing the index;
- Oracle searches index for the indexed column values accessed by the query
  - if the statement accesses only columns of the index, then data is read diretly from index, rather than from the table,
  - If a query accesses other columns in addition to the indexed columns, then Oracle finds the rows in the table by using a table access by rowid scan;
- **Index types:**
  - Index Unique Scans
  - Index Range Scans
  - Index Skip Scans
  - Index Full Scans
  - Fast Full Index Scans

Cost-based optimizers(CBO)

- Steps in **cost-based query optimization**
  1. Generate logically equivalent expressions using **equivalence rules**
  2. Annotate resultant expressions to get alternative query plans
  3. Choose the cheapest plan based on **estimated cost**

- Estimation of plan cost based on:
  - Statistical information about relations. Examples:
    - number of tuples, number of distinct values for an attribute, cardinality (# of tuples) in each table T, min and max values of each attribute A_j in T, Number of distinct values and selectivity

  - **Statistics estimation for intermediate results**
    - to compute cost of complex expressions

Cost-based optimizers(CBO)

- A plan tree consists of annotations at each node indicating:
  - The access methods to use for each relation
  - The implementation method to use for each operator

- A **Query Evaluation Plan** (**QEP**) defines exactly what algorithm is used for each operation, and how the execution of the operations is coordinated
  - To estimate the costs, all tables should be analysed.

    If there are no statistics in the data dictionary, the optimizer will guess them **based on the number of blocks allocated to the table.**
  - Important input data are:

    – Size of tables (number of rows, number of blocks)

    – Key constraints, foreign key constraints

    – Distribution of values for each attribute used in conditions (number of different values, minimal/maximal value).

Cost-based optimizers(CBO)

Estimated I/O cost of stratagies for Join Operations

STRATEGIES	COST
Block nested loop join	nBlocks(R) + (nBlocks(R) * nBlocks(S)), if buffer has only one block for R and S
	nBlocks(R) + [nBlocks(S)*(nBlocks(R)/(nBuffer − 2))], if (nBuffer − 2) blocks for R
	nBlocks(R) + nBlocks(S), if all blocks of R can be read into database buffer
Indexed nested loop join	Depends on indexing method; for example:
	nBlocks(R) + nTuples(R)*(nLevels$_A$ (I) + 1), if join attribute A in S is the primary key
	Blocks(R) + nTuples(R)*(nLevels$_A$ (I) + [SC$_A$ (R)/bFactor(R)]), for clustering index I on attribute A
Sort–merge join	nBlocks(R)*[$\log_2$ (nBlocks(R)] + nBlocks(S)*[$\log_2$ (nBlocks(S)], for sorts
	nBlocks(R) + nBlocks(S), for merge
Hash join	3(nBlocks(R) + nBlocks(S)), if hash index is held in memory
	2(nBlocks(R) + nBlocks(S))*[$\log_{nBuffer-1}$ (nBlocks(S)) − 1] + nBlocks(R) + nBlocks(S), otherwise

Cost-based optimizers(CBO)

Reserves (sid, bid, day, name)

Sailors (sib, name, rating, age)

Boats (bid, bname, colour)

Example

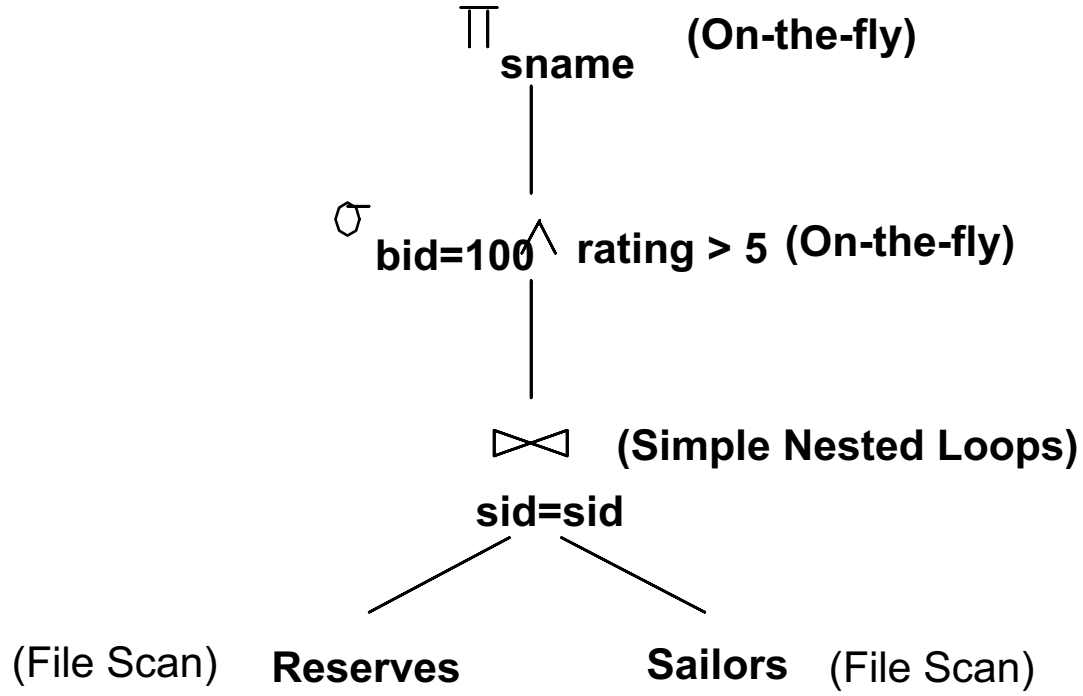
Query: List all the names of sailors with a rating of >5 who reserved the boat =100

Assumption:

- **Nr. Pages** of Reserves= **M=1000**, with100 **tuples/page**
- **Nr. Pages** of Sailors= **N=500**, with **80 tuples/page**
- Nr. **Reserved Boats** = **100** (**there are 10** different bid values **in the Reserves table**
- Nr. of values for the atribute rating in Sailors: 10 (**varies from 1 to 10**)
- Reserves **are uniform distribution** among the boats
- The number of sailors for each rating value is approximately the same (**uniform distribution of rating values among sailors**)
- Nr. Buffers free (pages)=5

Cost-based optimizers(CBO)

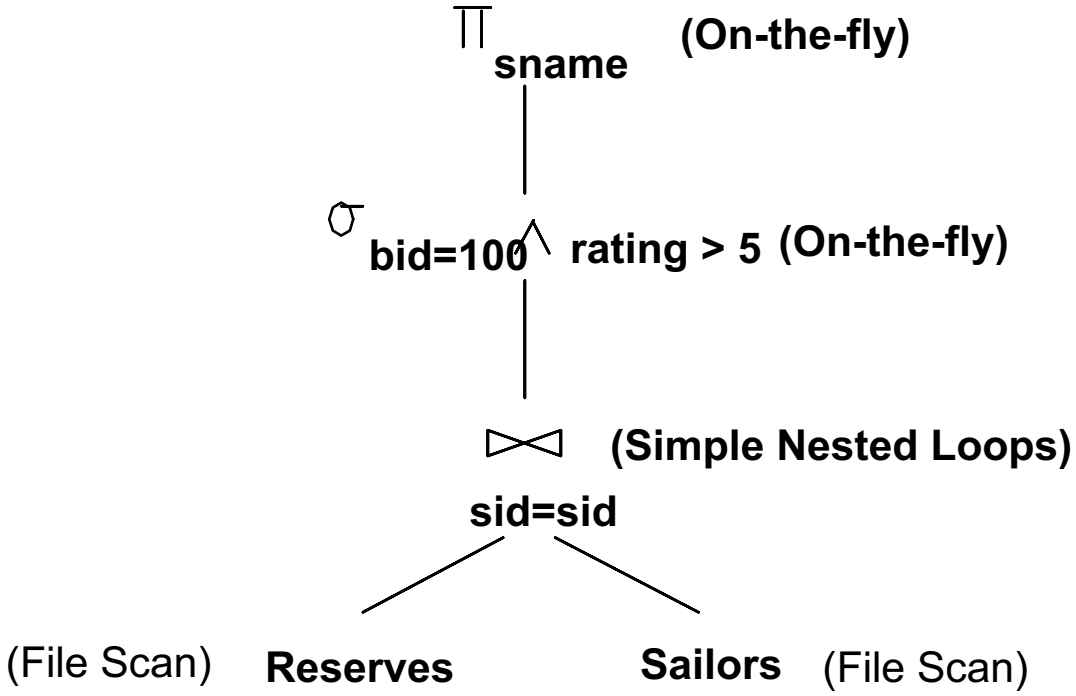
- **What is the I/O cost of the following evaluation plan?**



- ✓ The cost of the join is 1000 + 1000 * 500 = 501,000 I/Os (assuming page-oriented Simple NL join)
- ✓ The selection and projection are done on-the-fly; hence, do not incur additional I/Os

Cost-based optimizers(CBO)

- **What is the I/O cost of the following evaluation plan?**



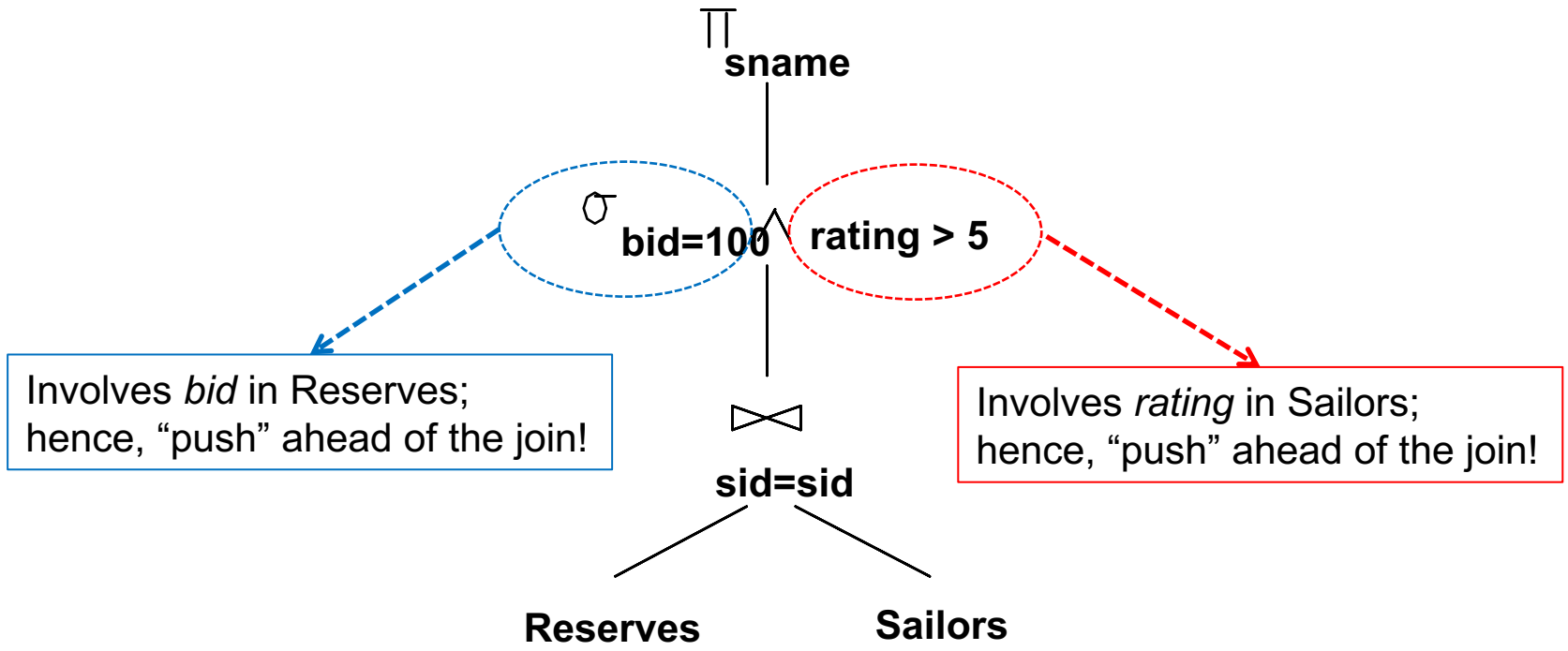
Nested Loops

$cost(R \bowtie S) = TR + TR * TS$

- ✓ The cost of the join is 1000 + 1000 * 500 = 501,000 I/Os (assuming page-oriented Simple NL join)
- ✓ The selection and projection are done on-the-fly; hence, do not incur additional I/Os

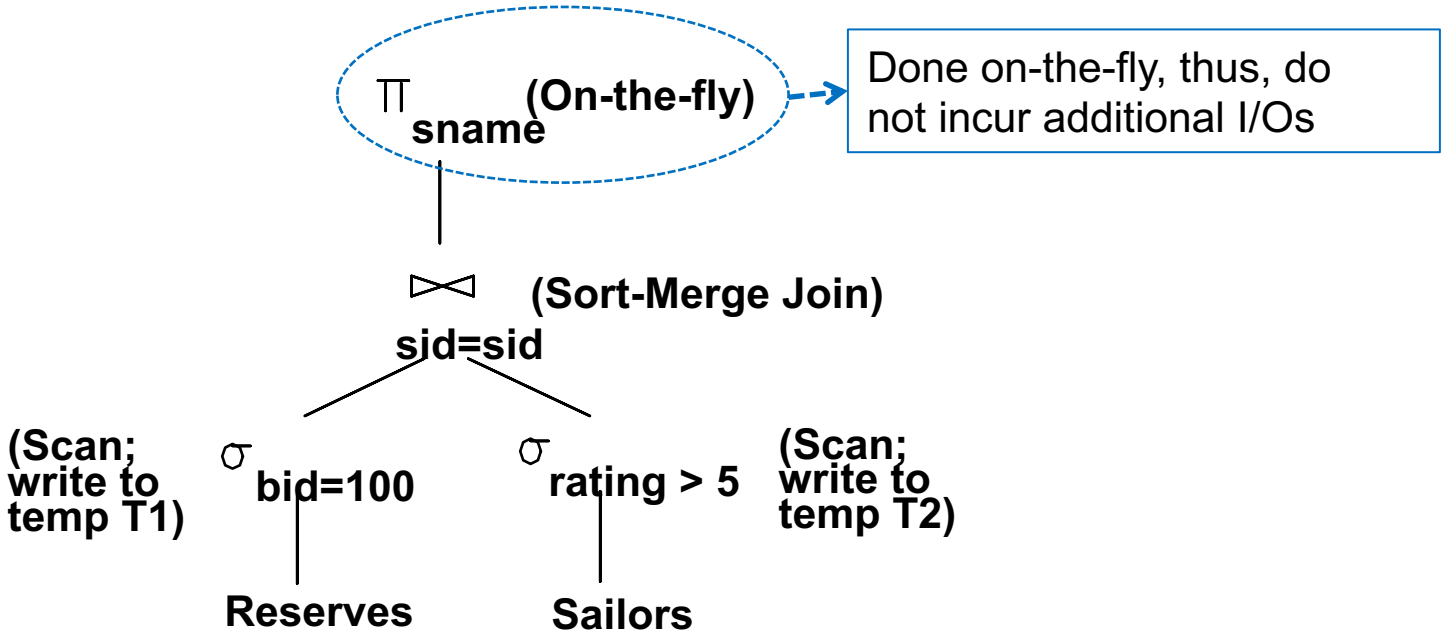
Cost-based optimizers(CBO)

- How can we reduce the cost of a join?
  - By reducing the sizes of the input relations!



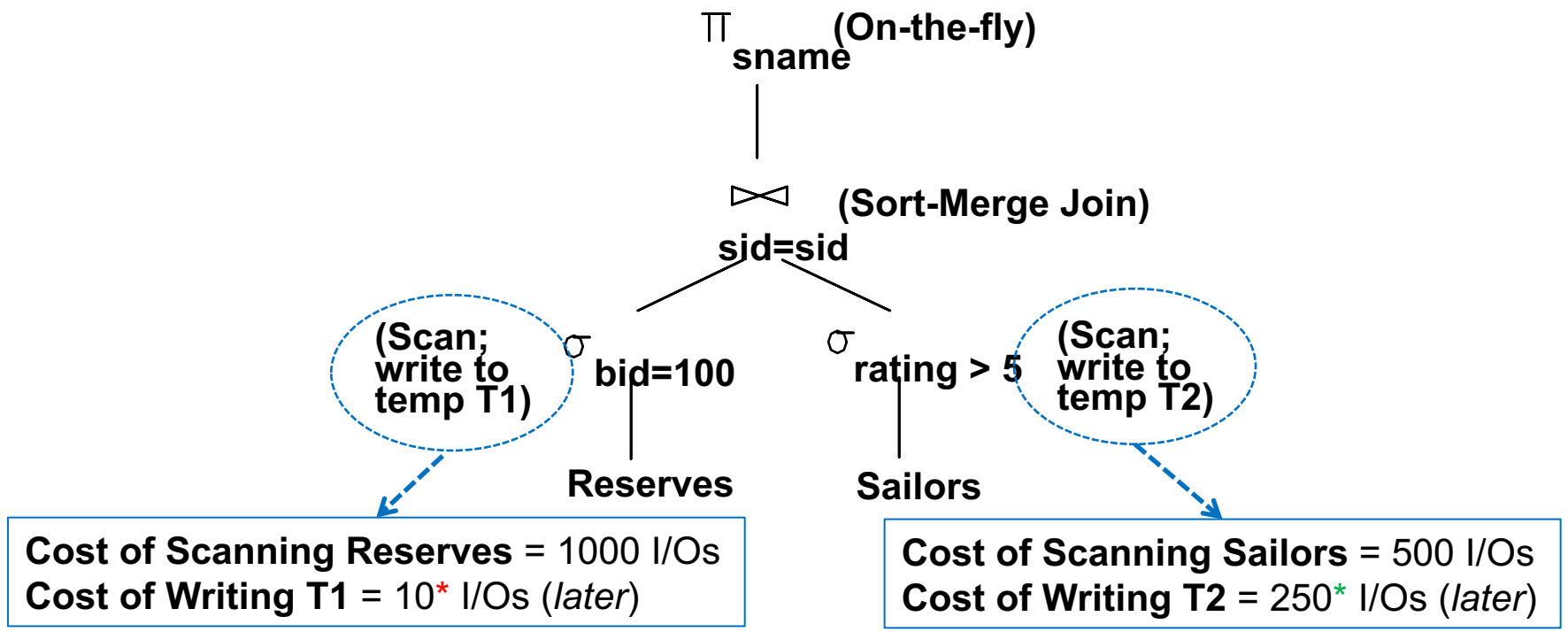
Cost-based optimizers(CBO)

- What is the I/O cost of the following evaluation plan?



Cost-based optimizers(CBO)

- What is the I/O cost of the following evaluation plan?

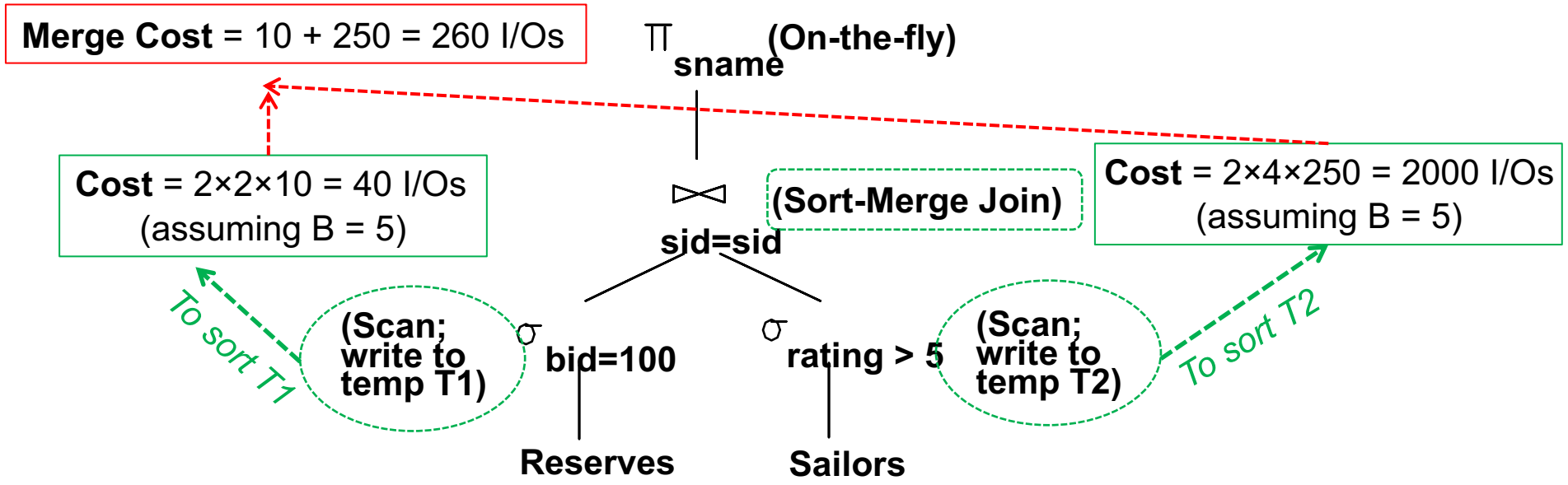


*Assuming 100 boats and uniform distribution of reservations across boats. → 1000/100

*Assuming 10 ratings and uniform distribution over ratings.

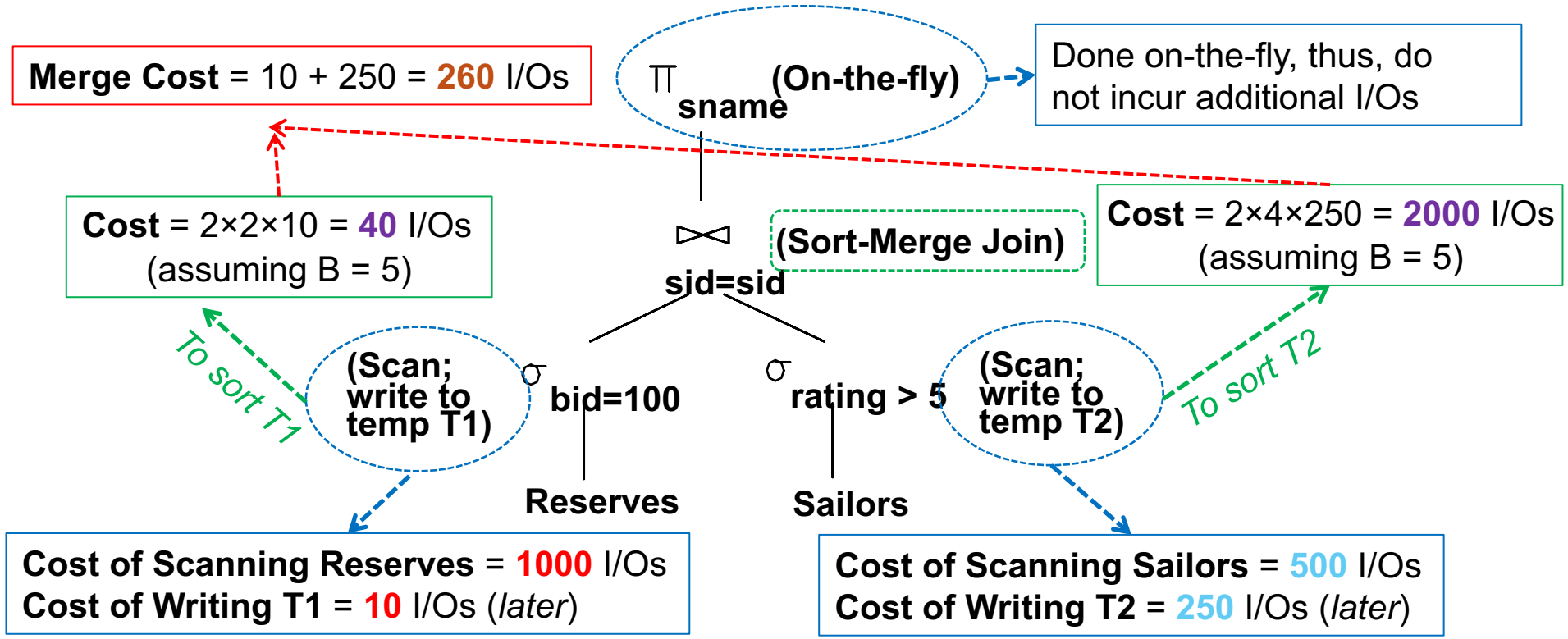
Cost-based optimizers(CBO)

- What is the I/O cost of the following evaluation plan?



Cost-based optimizers(CBO)

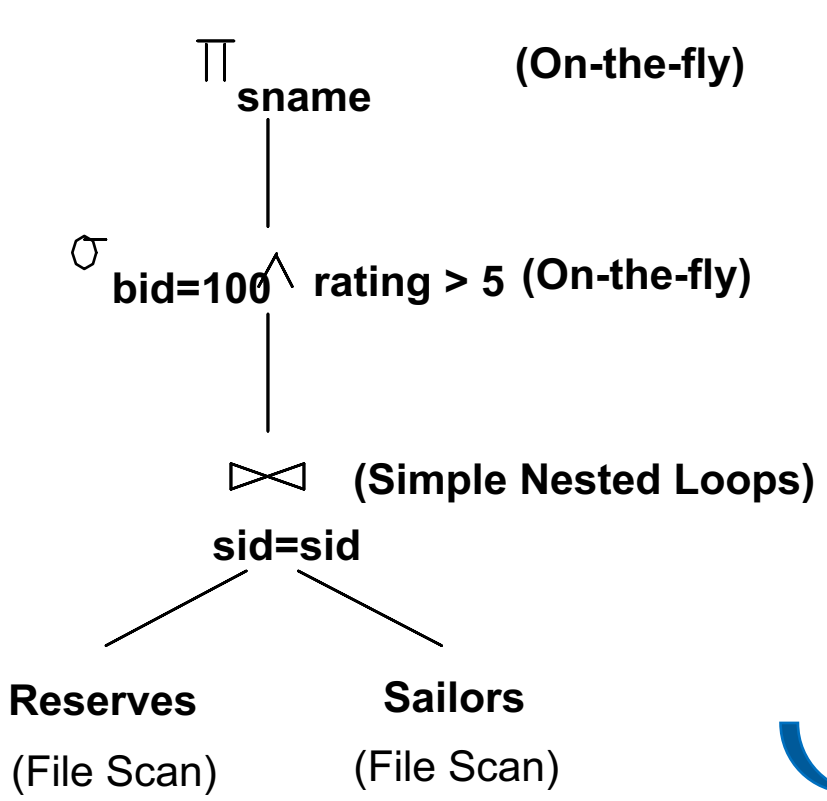
- What is the I/O cost of the following evaluation plan?

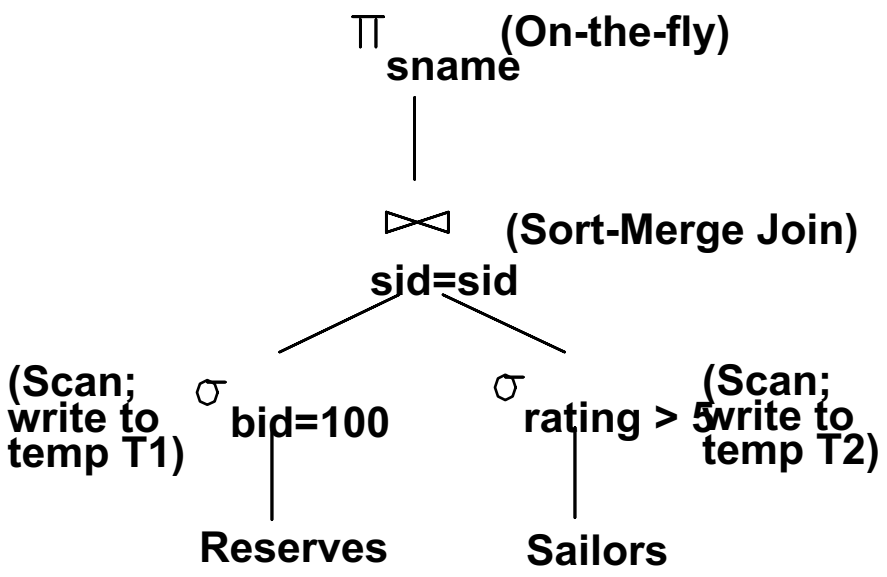


Total Cost = 1000 + 10 + 500 + 250 + 40 + 2000 + 260 = **4060** I/Os

Cost-based optimizers(CBO)

The I/O Costs of the *Two Q* Plans





Total Cost = 501, 000 I/Os

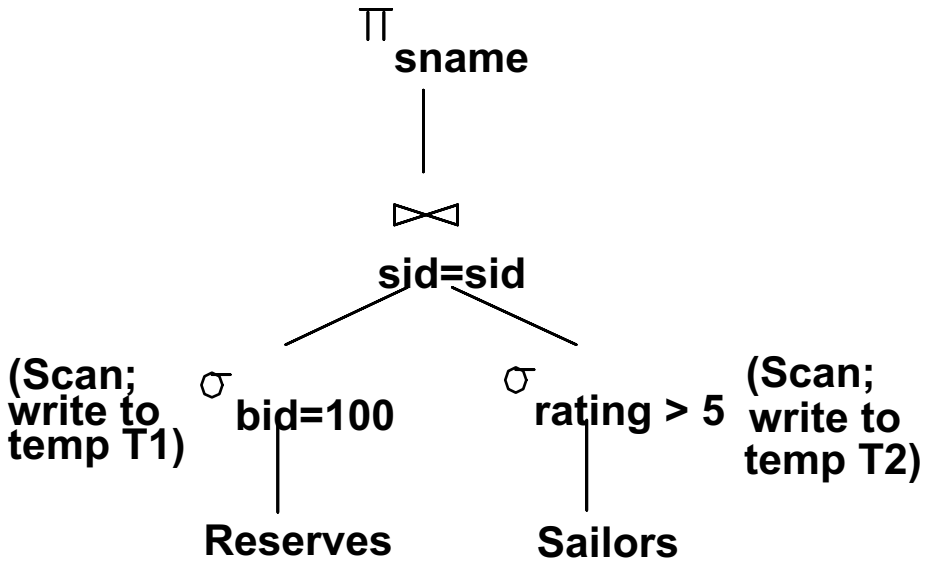
Total Cost = 4060 I/Os

Cost-based optimizers(CBO)

Pushing Projections

- How can we reduce the cost of a join?
  - By reducing the sizes of the input relations!

- Consider (again) the following plan:



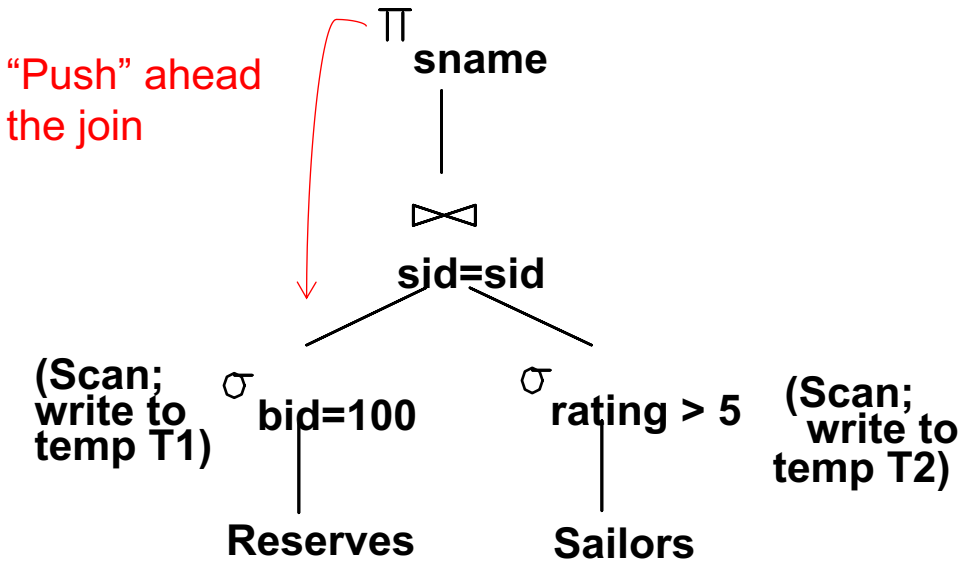
- What are the attributes required from T1 and T2?
  - *Sid* from T1
  - *Sid* and sname from T2

Hence, as we scan Reserves and Sailors, we can also remove unwanted columns (i.e., “Push” the projections ahead of the join)!

Cost-based optimizers(CBO)

- How can we reduce the cost of a join?
  - By reducing the sizes of the input relations!

- Consider (again) the following plan:



The cost after applying this heuristic can become 2000 I/Os (as opposed to 4060 I/Os with only pushing the selection)!

Query Optimizer

Executions plans

How to generate a plan

EXPLAIN PLAN command & dbms_xplan.display function

```
EXPLAIN PLAN FOR SELECT prod_name, avg(amount_sold)
                      FROM    sales s, products p
                      WHERE p.prod_id = s.prod_id
                      GROUP BY prod_name;

SELECT plan_table_output  FROM
table(dbms_xplan.display('plan_table',null,'basic'));
```

Understanding execution plans

Cardinality

- Estimate of number rows that will be returned by each operation
- Cardinality for a single column equality predicate = **total num of rows / num of distinct valu**es

Join Methods.

- **Nested loops joins** - For every row in the outer table, Oracle accesses all the rows in the inner table
  - **Useful** when **joining small subsets of d**ata and **not exists an efficient way to access the second table;**
- **Hash join - The smaller** of two **tables is scan** and resulting rows are **used to build a hash table on the join key in** memory.
  - The larger table is then scan, join column of the resulting rows are hashed and the values used to probe the hash table to find the matching rows.
  - Useful for larger tables & if we use aggregations functions

Understanding execution plans

Sort Merge Join - Consists of two steps:

1. Sort join operation: Both the inputs are sorted on the join key.
2. Merge join operation: The sorted lists are merged together.

   Useful when the join condition between two tables is **an inequality condition**

Cartesian Joins

- Joins every row from one data source with every row from the other data source, creating the Cartesian Product of the two sets.
  - Only good if tables **are very small.**
    - **choice if there** is no join condition specified in query

Outer Joins

- Returns all rows that satisfy the join condition and also returns all of the rows from the table for which no rows from the other table satisfy the join condition

Understanding execution plans

```
-------------------------------------------------------------------------------------------
| Id  | Operation                     | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |             |       |       |    12 (100)|          |
|   1 |  NESTED LOOPS                 |             |       |       |            |          |
|   2 |   NESTED LOOPS                |             |     1 |   211 |    12   (9)| 00:00:01 |
|   3 |    NESTED LOOPS               |             |     1 |   185 |    11  (10)| 00:00:01 |
|*  4 |     HASH JOIN                 |             |     1 |   155 |    10  (10)| 00:00:01 |
|   5 |      MERGE JOIN CARTESIAN     |             |   107 |  8774 |     6   (0)| 00:00:01 |
|*  6 |       TABLE ACCESS FULL       | DEPARTMENTS |     1 |    30 |     3   (0)| 00:00:01 |
|   7 |       BUFFER SORT             |             |   107 |  5564 |     3   (0)| 00:00:01 |
|   8 |        TABLE ACCESS FULL      | EMPLOYEES   |   107 |  5564 |     3   (0)| 00:00:01 |
|   9 |      TABLE ACCESS FULL        | EMPLOYEES   |   107 |  7811 |     3   (0)| 00:00:01 |
|* 10 |     TABLE ACCESS BY INDEX ROWID| DEPARTMENTS |     1 |    30 |     1   (0)| 00:00:01 |
|* 11 |      INDEX UNIQUE SCAN        | DEPT_ID_PK  |     1 |       |     0   (0)|          |
|* 12 |    INDEX UNIQUE SCAN          | JOB_ID_PK   |     1 |       |     0   (0)|          |
|  13 |   TABLE ACCESS BY INDEX ROWID | JOBS        |     1 |    26 |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------

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

   4 - access("E"."MANAGER_ID"="E"."EMPLOYEE_ID" AND
              "E"."DEPARTMENT_ID"="D"."DEPARTMENT_ID")
       filter("E"."SALARY"+("E"."SALARY"+"E"."COMMISSION_PCT
              ARY"+"E"."COMMISSION_PCT"))
   6 - filter("D"."DEPARTMENT_NAME"='Sales')
  10 - filter("D"."DEPARTMENT_NAME"='Sales')
  11 - access("E"."DEPARTMENT_ID"="D"."DEPARTMENT_ID")
  12 - access("E"."JOB_ID"="J"."JOB_ID")
```

Cardinality - estimated # of rows returned

Determine correct cardinality using a SELECT COUNT(*) from each table applying any WHERE Clause predicates belonging to that table

Understanding execution plans

Checking cardinality estimates

```
SELECT /*+ gather_plan_statistics */ p.prod_name, SUM(s.quantity_sold)
FROM        sales s, products p
WHERE       s.prod_id =p.prod_id GROUP By p.prod_name ;

SELECT * FROM table (
   DBMS_XPLAN.DISPLAY_CURSOR(FORMAT=>'ALLSTATS LAST'));
```

Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time	Buffers	OMem	1Mem	Used-Mem
0	SELECT STATEMENT		1		71	00:00:00.57	1638			
1	HASH GROUP BY		1	71	71	00:00:00.57	1638	799K	799K	3079K (0)
* 2	HASH JOIN		1	918K	918K	00:00:00.85	1638	933K	933K	1279K (0)
3	TABLE ACCESS STORAGE FULL	PRODUCTS	1	72	72	00:00:00.01	3			
4	PARTITION RANGE ALL		1	918K	918K	00:00:00.37	1635			
5	TABLE ACCESS STORAGE FULL	SALES	28	918K	918K	00:00:00.20	1635			

Estimated rows

Actual rows

Understanding execution plans

Solutions to incorrect cardinality estimates

Cause	Solution
Stale or missing statistics (not updated)	DBMS_STATS
Data Skew	Create a histogram
Multiple single column predicates on a table	Create a column group using DBMS_STATS.CREATE_EXTENDED_STATS
Function wrapped column	Create statistics on the function wrapped column using DBMS_STATS.CREATE_EXTENDED_STATS
Multiple columns used in a join	Create a column group on join columns using DBMS_STATS.CREATE_EXTENDED_STAT

Understanding execution plans

Access paths – Getting the data

Access Path	Explanation
Full table scan	Reads all rows from table & filters out those that do not meet the where clause predicates. Used when no index, DOP set etc
Table access by Rowid	Rowid specifies the datafile & data block containing the row and the location of the row in that block. Used if rowid supplied by index or in where clause
Index unique scan	Only one row will be returned. Used when stmt contains a UNIQUE or a PRIMARY KEY constraint that guarantees that only a single row is accessed
Index range scan	Accesses adjacent index entries returns ROWID values Used with equality on non-unique indexes or range predicate on unique index (<.>, between etc)

Understanding execution plans

Access paths – Getting the data

Access Path	Explanation
Full index scan	Processes all leaf blocks of an index, but only enough branch blocks to find 1st leaf block. Used when all necessary columns are in index & **order by clause matches index or if sort merge join is done**
Fast full index scan	when all necessary columns are in the index. Using multi-block IO
Index joins	Hash join of several indexes that together contain all the table columns that are referenced in the query. Won't eliminate a sort operation
Bitmap indexes	uses a bitmap for key values and a mapping function that converts each bit position to a rowid. Can efficiently merge indexes that correspond to several conditions in a WHERE clause

Understanding execution plans

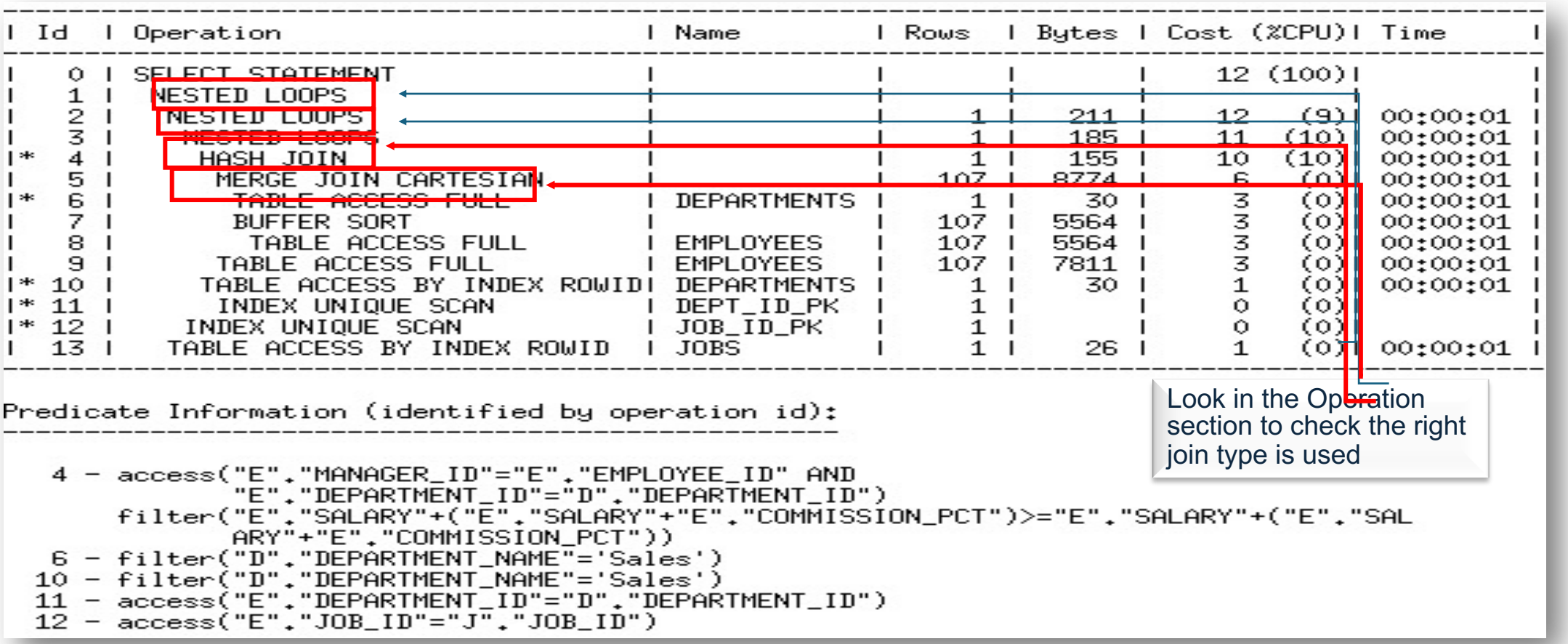
Identifying access paths **in an execution plan**

```
-----------------------------------------------------------------------------------------------
| Id  | Operation                     | Name            | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |                 |       |       |    12 (100)|          |
|   1 |  NESTED LOOPS                 |                 |       |       |            |          |
|   2 |   NESTED LOOPS                |                 |     1 |   211 |    12   (9)| 00:00:01 |
|   3 |    NESTED LOOPS               |                 |     1 |   185 |    11  (10)| 00:00:01 |
|*  4 |     HASH JOIN                 |                 |     1 |   155 |    10  (10)| 00:00:01 |
|   5 |      MERGE JOIN CARTESIAN     |                 |   107 |  8774 |     6   (0)| 00:00:01 |
|*  6 |       TABLE ACCESS FULL       | DEPARTMENTS     |     1 |    30 |     2   (0)| 00:00:01 |
|   7 |       BUFFER SORT             |                 |   107 |  5564 |     3   (0)| 00:00:01 |
|   8 |        TABLE ACCESS FULL      | EMPLOYEES       |   107 |  5564 |     3   (0)| 00:00:01 |
|   9 |      TABLE ACCESS FULL        | EMPLOYEES       |   107 |  7811 |     3   (0)| 00:00:01 |
|* 10 |     TABLE ACCESS BY INDEX ROWID| DEPARTMENTS    |     1 |    30 |     1   (0)| 00:00:01 |
|* 11 |      INDEX UNIQUE SCAN        | DEPT_ID_PK      |     1 |       |     0   (0)|          |
|* 12 |    INDEX UNIQUE SCAN          | JOB_ID_PK       |     1 |       |     0   (0)|          |
|  13 |   TABLE ACCESS BY INDEX ROWID | JOBS            |     1 |    26 |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------

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

   4 - access("E"."MANAGER_ID"="E"."EMPLOYEE_ID" AND
              "E"."DEPARTMENT_ID"="D"."DEPARTMENT_ID")
       filter("E"."SALARY"+("E"."SALARY"+"E"."COMMISSION_PCT")>="E"."SALARY"+("E"."SAL
              ARY"+"E"."COMMISSION_PCT"))
   6 - filter("D"."DEPARTMENT_NAME"='Sales')
  10 - filter("D"."DEPARTMENT_NAME"='Sales')
  11 - access("E"."DEPARTMENT_ID"="D"."DEPARTMENT_ID")
  12 - access("E"."JOB_ID"="J"."JOB_ID")
```

Look in Operation section to see how an object is being accessed

Understanding execution plans

Identifying join methods in an execution plan



```
-------------------------------------------------------------------------------------------
| Id  | Operation                      | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT               |             |       |       |    12 (100)|          |
|   1 |  NESTED LOOPS                  |             |       |       |            |          |
|   2 |   NESTED LOOPS                 |             |     1 |   211 |    12   (9)| 00:00:01 |
|   3 |    NESTED LOOPS                |             |     1 |   185 |    11  (10)| 00:00:01 |
|*  4 |     HASH JOIN                  |             |     1 |   155 |    10  (10)| 00:00:01 |
|   5 |      MERGE JOIN CARTESIAN      |             |   107 |  8774 |     6   (0)| 00:00:01 |
|*  6 |       TABLE ACCESS FULL        | DEPARTMENTS |     1 |    30 |     3   (0)| 00:00:01 |
|   7 |       BUFFER SORT              |             |   107 |  5564 |     3   (0)| 00:00:01 |
|   8 |        TABLE ACCESS FULL       | EMPLOYEES   |   107 |  5564 |     3   (0)| 00:00:01 |
|   9 |      TABLE ACCESS FULL         | EMPLOYEES   |   107 |  7811 |     3   (0)| 00:00:01 |
|* 10 |     TABLE ACCESS BY INDEX ROWID| DEPARTMENTS |     1 |    30 |     1   (0)| 00:00:01 |
|* 11 |      INDEX UNIQUE SCAN         | DEPT_ID_PK  |     1 |       |     0   (0)|          |
|* 12 |    INDEX UNIQUE SCAN           | JOB_ID_PK   |     1 |       |     0   (0)|          |
|  13 |   TABLE ACCESS BY INDEX ROWID  | JOBS        |     1 |    26 |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------

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

   4 - access("E"."MANAGER_ID"="E"."EMPLOYEE_ID" AND
              "E"."DEPARTMENT_ID"="D"."DEPARTMENT_ID")
       filter("E"."SALARY"+("E"."SALARY"+"E"."COMMISSION_PCT")>="E"."SALARY"+("E"."SAL
              ARY"+"E"."COMMISSION_PCT"))
   6 - filter("D"."DEPARTMENT_NAME"='Sales')
  10 - filter("D"."DEPARTMENT_NAME"='Sales')
  11 - access("E"."DEPARTMENT_ID"="D"."DEPARTMENT_ID")
  12 - access("E"."JOB_ID"="J"."JOB_ID")
```

Look in the Operation section to check the right join type is used

Understanding execution plans

Identifying join order in an execution plan

```
-------------------------------------------------------------------------------------------
| Id  | Operation                     | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |             |       |       |    12 (100)|          |
|   1 |  NESTED LOOPS                 |             |       |       |            |          |
|   2 |   NESTED LOOPS                |             |     1 |   211 |    12   (9)| 00:00:01 |
|   3 |    NESTED LOOPS               |             |     1 |   185 |    11  (10)| 00:00:01 |
|*  4 |     HASH JOIN                 |             |     1 |   155 |    10  (10)| 00:00:01 |
|   5 |      MERGE JOIN CARTESIAN     |             |   107 |  8774 |     6   (0)| 00:00:01 |
|*  6 |       1TABLE ACCESS FULL      | DEPARTMENTS |     1 |    30 |     3   (0)| 00:00:01 |
|   7 |        BUFFER SORT            |             |   107 |  5564 |     3   (0)| 00:00:01 |
|   8 |       2TABLE ACCESS FULL      | EMPLOYEES   |   107 |  5564 |     3   (0)| 00:00:01 |
|   9 |     3TABLE ACCESS FULL        | EMPLOYEES   |   107 |  7811 |     3   (0)| 00:00:01 |
|* 10 |    4 TABLE ACCESS BY INDEX ROWID| DEPARTMENTS |     1 |    30 |     1   (0)| 00:00:01 |
|* 11 |       INDEX UNIQUE SCAN       | DEPT_ID_PK  |     1 |       |     0   (0)|          |
|* 12 |     INDEX UNIQUE SCAN         | JOB_ID_PK   |     1 |       |     0   (0)|          |
|  13 |   5TABLE ACCESS BY INDEX ROWID| JOBS        |     1 |    26 |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------

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

   4 - access("E"."MANAGER_ID"="E"."EMPLOYEE_ID" AND
              "E"."DEPARTMENT_ID"="D"."DEPARTMENT_ID")
       filter("E"."SALARY"+("E"."SALARY"+"E"."COMMISSION_PCT")>="E"."SALARY"+("E"."SAL
              ARY"+"E"."COMMISSION_PCT"))
   6 - filter("D"."DEPARTMENT_NAME"='Sales')
  10 - filter("D"."DEPARTMENT_NAME"='Sales')
  11 - access("E"."DEPARTMENT_ID"="D"."DEPARTMENT_ID")
  12 - access("E"."JOB_ID"="J"."JOB_ID")
```

start with the table that reduce the result set the most

If the join order is not correct, check the statistics, cardinality & access methods

Understanding execution plans

Finding the join order for complex SQL

- It can be hard to determine Join Order for Complex SQL statements but it is easily visible in the outline data of plan

SELECT * FROM table(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$5428C7F1")
      MERGE(@"SEL$2")
      MERGE(@"SEL$3")
      OUTLINE(@"SEL$1")
      OUTLINE(@"SEL$2")
      OUTLINE(@"SEL$3")
      FULL(@"SEL$5428C7F1" "D"@"SEL$3")
      INDEX_RS_ASC(@"SEL$5428C7F1" "E"@"SEL$3" ("EMPLOYEES"."DEPARTMENT_ID"))
      INDEX_RS_ASC(@"SEL$5428C7F1" "E"@"SEL$2" ("EMPLOYEES"."MANAGER_ID"))
      INDEX_RS_ASC(@"SEL$5428C7F1" "J"@"SEL$2" ("JOB
      INDEX(@"SEL$5428C7F1" "D"@"SEL$2" ("DEPARTMENT
      LEADING(@"SEL$5428C7F1" "D"@"SEL$3" "E"@"SEL$3
      USE_NL(@"SEL$5428C7F1" "E"@"SEL$3")
```

The leading hint tells you the join order

Causes wrong join order

- Incorrect single table cardinality estimates
- Incorrect join cardinality estimates

Order of execution of the operations

We basically have 4 rules:

1) Following the operation id=0, the first operation to be executed in the plan will be the most indented one that does not have a daughter.
2) The sister operation of this operation will then be executed if it exists and does not have a child operation.
3) If the sister operation has a child, we will continue checking the operations that descend from it until we find an operation that does not have a child, this will be the next operation to be executed.
4) After executing an operation, if it has no siblings, the next operation to be executed will be the parent operation.

```
Plan hash value: 4239905139

-------------------------------------------------------------------------------
| Id  | Pid | Ord | Operation                | Name        | Rows  | By
-------------------------------------------------------------------------------
|   0 |     |   6 | SELECT STATEMENT         |             |       |
|   1 |   0 |   5 |  SORT GROUP BY           |             |    11 |
|   2 |   1 |   4 |   MERGE JOIN CARTESIAN   |             |  2889 | 20
|   3 |   2 |   1 |    TABLE ACCESS FULL     | EMPLOYEES   |   107 |
|   4 |   2 |   3 |    BUFFER SORT           |             |    27 |
|   5 |   4 |   2 |     INDEX FAST FULL SCAN | DEPT_ID_PK  |    27 |
-------------------------------------------------------------------------------
```

Order of execution of the operations

Try to do

```
Plan hash value: 2215075747

-------------------------------------------------------------------------------------------
| Id  | Operation                | Name          | Rows  | Bytes | Cost (%CPU)| Time
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT         |               |       |       |   410 (100)|
|*  1 |  FILTER                  |               |       |       |            |
|   2 |   SORT GROUP BY          |               |    20 |   820 |   410   (1)| 00:00:05
|*  3 |    HASH JOIN             |               | 55500 |  2222K|   408   (1)| 00:00:05
|   4 |     TABLE ACCESS FULL    | COUNTRIES     |    23 |   345 |     3   (0)| 00:00:01
|   5 |     TABLE ACCESS FULL    | CUSTOMERS     | 55500 |  1409K|   405   (1)| 00:00:05
|   6 |   SORT AGGREGATE         |               |     1 |    10 |            |
|*  7 |    HASH JOIN             |               | 55500 |   541K|   406   (1)| 00:00:05
|   8 |     INDEX FULL SCAN      | COUNTRIES_PK  |    23 |   115 |     1   (0)| 00:00:01
|   9 |     TABLE ACCESS FULL    | CUSTOMERS     | 55500 |   270K|   405   (1)| 00:00:05
|  10 |   SORT GROUP BY          |               |     1 |    13 |            |
|  11 |    VIEW                  |               |    12 |   156 |   407   (1)| 00:00:05
|  12 |     SORT GROUP BY        |               |    12 |   528 |   407   (1)| 00:00:05
|  13 |      NESTED LOOPS        |               |   162 |  7128 |   407   (1)| 00:00:05
|  14 |       VIEW               | VW_GBF_8      |   162 |  6318 |   407   (1)| 00:00:05
|  15 |        SORT GROUP BY     |               |   162 |  4212 |   407   (1)| 00:00:05
|  16 |         TABLE ACCESS FULL| CUSTOMERS     | 55500 |  1409K|   405   (1)| 00:00:05
|* 17 |       INDEX UNIQUE SCAN  | COUNTRIES_PK  |     1 |     5 |     0   (0)|
-------------------------------------------------------------------------------------------
```

1) first operation -the most indented one that does not have a daughter.
2) The sister operation -if it exists and does not have a child operation.
3) If the sister operation has a child, - descend from it until we find an operation that does not have a child,
4) After, if it has no siblings, execute the parent operation.

Order of execution of the operations

SOLUTION

```
Plan hash value: 2215075747

-----------------------------------------------------------------------------------------
| Id  | Pid | Ord | Operation                | Name         | Rows  | Bytes | Cost
-----------------------------------------------------------------------------------------
|   0 |     |  18 | SELECT STATEMENT         |              |       |       |   41
|*  1 |   0 |  17 |  FILTER                  |              |       |       |
|   2 |   1 |   4 |   SORT GROUP BY          |              |    20 |   820 |   41
|*  3 |   2 |   3 |    HASH JOIN             |              | 55500 |  2222K|   40
|   4 |   3 |   1 |     TABLE ACCESS FULL    | COUNTRIES    |    23 |   345 |
|   5 |   3 |   2 |     TABLE ACCESS FULL    | CUSTOMERS    | 55500 |  1409K|   405
|   6 |   1 |   8 |   SORT AGGREGATE         |              |     1 |    10 |
|*  7 |   6 |   7 |    HASH JOIN             |              | 55500 |   541K|   406
|   8 |   7 |   5 |     INDEX FULL SCAN      | COUNTRIES_PK |    23 |   115 |     1
|   9 |   7 |   6 |     TABLE ACCESS FULL    | CUSTOMERS    | 55500 |   270K|   405
|  10 |   1 |  16 |   SORT GROUP BY          |              |     1 |    13 |
|  11 |  10 |  15 |    VIEW                  |              |    12 |   156 |   407
|  12 |  11 |  14 |     SORT GROUP BY        |              |    12 |   528 |   407
|  13 |  12 |  13 |      NESTED LOOPS        |              |   162 |  7128 |   407
|  14 |  13 |  11 |       VIEW               | VW_GBF_8     |   162 |  6318 |   407
|  15 |  14 |  10 |        SORT GROUP BY     |              |   162 |  4212 |   407
|  16 |  15 |   9 |         TABLE ACCESS FULL| CUSTOMERS    | 55500 |  1409K|   405
|* 17 |  13 |  12 |       INDEX UNIQUE SCAN  | COUNTRIES_PK |     1 |     5 |     0
-----------------------------------------------------------------------------------------
```

1) first operation -the most indented one that does not have a daughter.
2) The sister operation -if it exists and does not have a child operation.
3) If the sister operation has a child, - descend from it until we find an operation that does not have a child,
4) After, if it has no siblings, execute the parent operation.

References

Slides based on

- https://beginner-sql-tutorial.com/sql-query-tuning.htm;
- Database Systems A Practical Approach to Design, implementation, Management, Thomas Connolly, Carolyn Begg;
- The Oracle Optimizer Explain the Explain Plan
- https://blogs.oracle.com/optimizer/
- https://www.oracle.com/pt/database/technologies/