# 10. Query Optimizer

## Intro

It's in charge of selecting and generating an efficient strategy for query execution(query execution plan).
It also guarantees the **data independency property**, that is, the way a SQL query is written does not affect the way in which it is implemented and viceversa, a physical reorganization of data does not require rewriting SQL queries.
There are several steps to the overall optimization.

### Overview of Lexical, Syntactic and Semantic analysis

In this step, the **optimizer** checks for any **lexical**,**syntactic** or **semantic** errors.
Once the analysis is done, it translates the query into an internal procedural representation in *extended* **relational algebra**.

### Overview of Algebraic optimization

A set of transformations that are considered to be always beneficial, such as moving the selection before the join.
This step is usually independent of the data distribuction.
In this step, a **query tree** in canonical form is outputd.

### Overview of Cost based optimization

Selection of the best execution plan by evaluating the **execution cost.**
It decides the selection of:

- **access methods** for each table
- **algorithm** for each relational operator.
  An **executable** is generated at the end of this step.

### Overview of Execution modes

Two different modes

#### Compile and go

Compilation and immediate execution of the statement, without storing it. No dependencies are needed

### Compile and store

Compilation and storing it in the database together with its dependencies.
It's executen then **on demand.**
Whenever the data structure changes, a recompilation is needed

## Algebraic Optimization

It is based on equivalence transformations, that is, two relational expressions are equivalent if they produce the same result.
There are several transformations

### Atomization of selection

A conjuction(AND) of selection can be transformed in nested selections.

$$\sigma_{F1 \wedge F2}(E) \equiv \sigma_{F2}(\sigma_{F1}(E)) \equiv \sigma_{F1}(\sigma_{F2}(E))$$

### Cascading projections

Almost the same concept of the **atomization of selection**

$$\pi_X(E) \equiv \pi_X(\pi_{X,Y}(E))$$

### Anticipation of selection with respect to join

A selection on a join can be moved so that the selection applies only on the single table.
That way, we reduce the size of the join.

$$\sigma_F(E_1 \bowtie E_2) \equiv E_1 \bowtie (\sigma_F(E_2))$$

*F is a predicate on attributes of the $E_2$ tables

### Anticipation of projection with respect to join

Almost the same concept of **Anticipation of selection with respect to join**.
Here, **L1** is the subset of **L** without the attributes of E2, that is **L1=L-Schema(E2)**; same thing goes for **L2**.
Moreover, **J** is the set of attributes needed to evaluate the **join predicate** $p$
It's used to reduce the size of the intermediate result

$$\pi_L(E_1 \bowtie_p E_2) \equiv \pi_L((\pi_{L1,J}(E_1)) \bowtie_p (\pi_{L2,J}(E_2)))$$

### Cartesian into join

This is obvious since the definition of join is the application of a selection on a cartesian product.
Here, predicate $F$ only relates attributes in both E1 and E2.

$$\sigma_F (E_1 \times E_2) \equiv E_1 \bowtie_F E_2$$

## Distribution of selection with respect to ..

- **Union:**

$$\sigma_F(E_1 \cup E_2) \equiv (\sigma_F (E_1)) \cup (\sigma_F (E_2))$$

- **Difference:**

$$\begin{aligned}\sigma_F(E_1 - E_2) &\equiv (\sigma_F (E_1)) - (\sigma_F (E_2)) \\ &\equiv (\sigma_F (E_1)) - E_2\end{aligned}$$

## Distribution of projection with respect to union

It reduces the size of intermediate results.

$$\pi_X(E_1 \cup E_2) \equiv (\pi_X(E_1)) \cup (\pi_X(E_2))$$

## Distribution of join with respect to union

$$E \bowtie (E_1 \cup E_2) \equiv (E \bowtie E_1) \cup (E \bowtie E_2)$$

# Cost based optimization

It's based on **data profiles** and approximate cost formulas for **access operations**

## Data profiles

They are basically statistics about table size and data distribution among them

### Table profiles

Quantitative information on the characteristics of tables and columns, such as:

- **cardinality** in each table
- **size in bytes** of tuples in a table
- number of **distinct values** for each attribute in a given table

- **min** and **max** values for each attribute in a given table
  These stats influence decisions on DB scan or using an index to access data.
  For example, we can use table profiles to estimate the size of intermediate relation expressions.

$$\text{Card}\,(\sigma_{Ai\,=\,v}\,(T)) \approx \text{Card}\,(T)/\,\text{Val}\,(A_i \text{ in } T)$$

Here, we can estimate the size of a selection.
*Val* is the number of distinct values of $A_i$ in $T$.
This formula is based under the hypothesis of uniform distribution

## Access operations

In a query tree, leaves represent physical structures such as indexes etc.
Intermediates nodes are instead operations on data.

### Sequential scan

The DBMS executes sequential access to all tuples in a table. It's also called **full table scan**.
During the scan, certain operations are performed:

- **projection**
- **selection**
- **sorting**(locally inside each block)
- **insert/update/delete**(locally inside each block)
  This approach is performed when there are no other options available(no indexes available or using an index wouldn't reduce I/O cost)

### Sorting

Classical algorithms are used to perform the sorting.
It uses the secondary memory to store intermediate results.

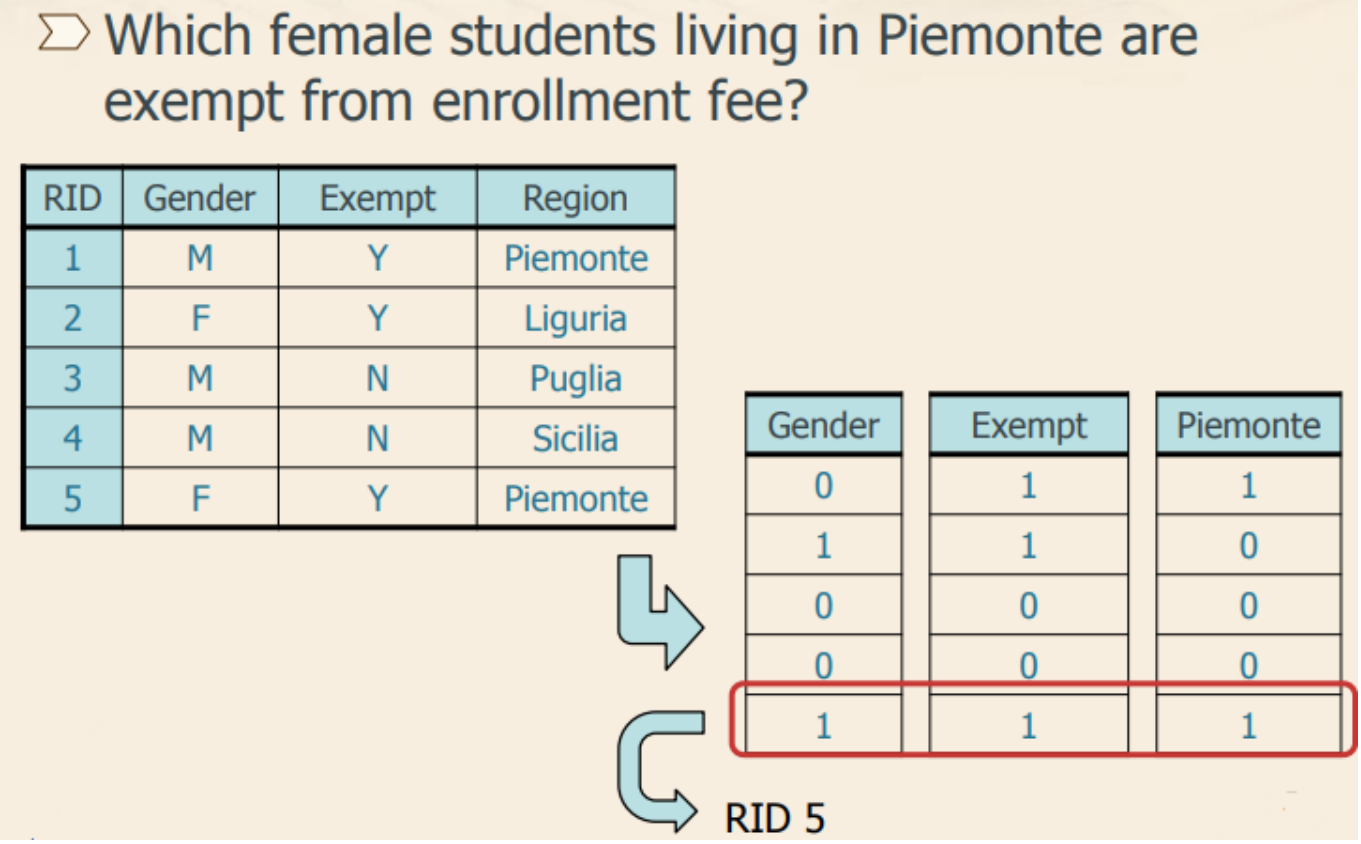
### Predicate valuation

Index access can be exploited if available.
For equality predicate, an **hash**,**b+tree** or bitmap indexes are used
For range predicate, only **b+tree** can be used as it's the only useful.
Instead, for predicates with *limited* selectivity,alas most of the tuples satisfy the predicate, a full table scan is better(or even bitmap can be considered).

In case of a *conjuction of predicates*, the **most selective** predicate is evaluated first. If an index is available, it will be used to read data. Then the other predicates are evaluated on the

intermediate results.
If a bitmap index is available on both predicates, the system can optimize by computing the **intersection** of **RIDs**.

If we have a **disjunction of predicates**, indexes can be exploited only if all predicates are supported by an index.




*Example of optimization based on the exploiting of bitmap indexes*

### Join operations

Joins are critical operations, especially in a relational dbms.
There are several algorithms to perform these operations.

### Nested loop

A single full scan is done on the outer table and fo reach tuple, a full scan on the inner table is performed.
As the name implies, it's a double nested for loop.
It's efficient when the inner table is **small** and the join attribute is **indexed**.
It's **asymmetric** as it depends on the **inner table.**

### Merge scan

Both tables are firstly sorted on the join attributes.
Then they are both scanned in parallel.

It's **symmetric** however requires sorting both tables, even though they might be already sorted by previous operations and stored on disk.

### Hash Join

An hash function is applied on join attributes of both tables.
Tuples to be joined end up in same hash buckets.
A local sort and join is performed into each bucket.
It's a **very fast** join technique

### Bitmapped join index

A bitmap matrix is exploited to perform a join operations.
Basically, a bitmap matrix precomputes the join between A and B:

- one column for each **RID** in table **A**
- one row for each **RID** in table **B**
  A **1** in a cell indicates that $A_x$ and $B_y$ will be joined up, **0** otherwise.
  Updates may be slow and it's typically used in OLAP queries where large tables must be joined up.

| RID | 1 | 2 | ... | n |
|---|---|---|---|---|
| 1 | 0 | 0 | ... | 1 |
| 2 | 0 | 1 | ... | 0 |
| 3 | 0 | 0 | ... | 1 |
| 4 | 1 | 0 | ... | 0 |
| ... | ... | ... | ... | 0 |

*RID 4 of table A and RID 1 of table B can be joined.*

### Group by

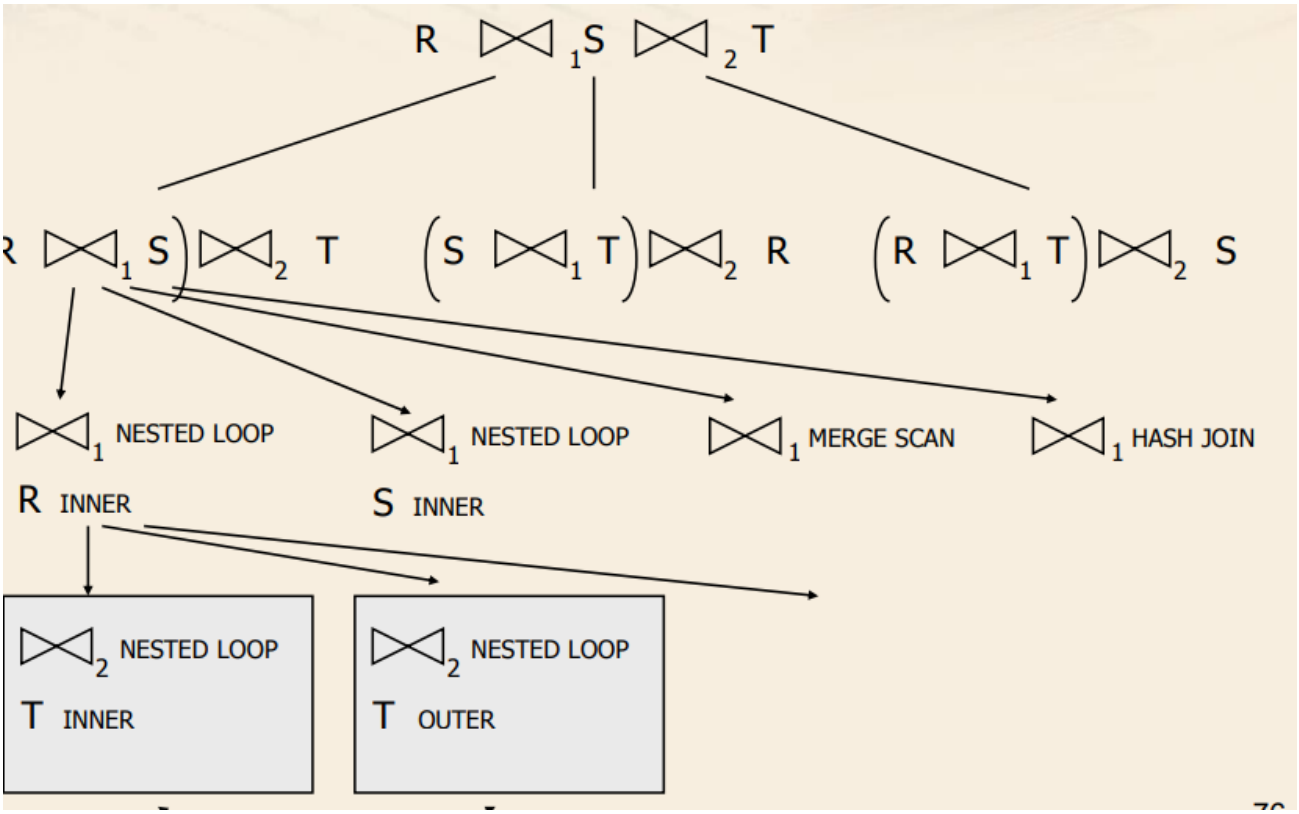
Can be performed in two ways:

- **Sort based:**
  1. Firstly, sort the resulting table on the group by attributes
  2. Then compute aggregate functions on groups
- **Hash based:**
  1. Firstly **hash functions** are applied on the group by attributes
  2. Next, each bucket is sorted and aggregation functions are computed
  **Materialized views** can be exploited to improve performance.

# Execution plan selection

It's done by evaluating the cost of different alternatives for **reading each table** and **executing each relation operator**.
Several criterias are considered:

- The way **data** is **read** from **disk**
  - full scan, index etc
- **Execution order** aong operators
  - jon order between two join operations
- **Algorithms** used to implement each operator
  - join algorithms etc.
- When to perform sort if needed.
  The optimizer builds a *tree of alternatives*.




Here, there are 4 join methods for each join order for both joins, thus $4 \times 4 \times 3 = 48$ alternatives.

Then the optimizer choses the leaf node(the blue rectangle) with the lowest cost, computed, by means of *operation research* techniques, as follows:

$$C_{Total} = C_{I/O} \times n_{I/O} + C_{cpu} \times n_{cpu}$$

- $n_{I/O}$ is the number of I/O operations
- $n_{cpu}$ is the number of CPU operations