CHAPTER III: RELATIONAL ALGEBRA

1. DEFINITION

Relational algebra is a branch of mathematical algebra that was developed in the context of relational databases. It was introduced by **Edgar F. Codd** in the 1970s. Relational algebra provides a formal set of operations for manipulating data stored in relational tables. These operations form the foundation of **SQL (Structured Query Language)**, which is widely used to interact with relational databases.

In practice, it is possible to create highly complex algebraic expressions to perform all relational queries using a limited set of fundamental operations. These basic operations include:

- **Selection**, denoted by **σ**
- **Projection**, denoted by **π**

} **Unary operators**

- **Union**, represented by **∪**
- **Intersection**, denoted by ∩
- **Difference**, denoted by **-**
- **Cartesian product**, denoted by ×
- **Division**, denoted by ÷
- **Join**, denoted by ⋈

} **Set operators**

The first two operations (**selection** and **projection**) are **unary operators** (as they take a single relation as input), while the others are **set operators** (as they take two relations as input).

2. UNARY OPERATORS

In relational algebra, **unary operations** are operations that act on a single relation (table). These operations allow filtering, transforming, or selecting data from a single table.

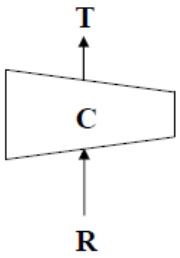
2.1. Restriction (σ - Sigma) (Also Called Selection)

This operation is applied to a relation **RELATION1**, producing a new relation **RELATION2** with the same schema but containing only the tuples that satisfy the specified condition. In other words, **restriction (selection)** allows retrieving rows that meet a specific condition.

Possible conditions follow the format:
<Attribute> <Operator> <Value>,
where the operator is a comparison operator chosen from:
{=, <, ≤, ≥, >, ≠} or a logical operator (**AND, OR, NOT**).

The attribute must belong to the relation on which the condition is applied.
The **restriction operation** can be represented using **textual and graphical** formalism.

Selection can be represented using different notations:



- **T = σ Condition (R)** *(most commonly used)*
- **T = SELECTION (R, condition)**
- **T = RESTRICT (R / Condition)**

Example:

Consider the following **Employee** relation:

EmpID	LastName	FirstName	Salary	Department
1	Khaldi	Samir	65000	Finance
2	Haroun	Ali	60000	HR (DRH)
3	Melzi	Aicha	45000	Finance
4	Heddad	Kawla	52000	HR (DRH)
5	Batli	Youssef	38000	Sales

To select employees earning **more than 50,000 DA**, we can use any of the following expressions:

- **R1 = σ Salaire> 50000 (Employe)**
- **R1 = SELECTION (Employe, Salary > 50000)**
- **R1 = RESTRICT (Employe / Salary > 50000)**

Resulting Relation R1:

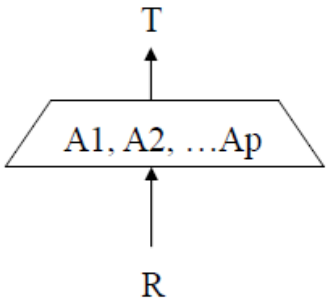
EmpID	LastName	FirstName	Salary	Department
1	Khaldi	Samir	65000	Finance
2	Haroun	Ali	60000	HR (DRH)
4	Heddad	Kawla	52000	HR (DRH)

2.2. Projection (π - Pi)

Projection is an operation on a relation **RELATION1** that creates a new relation **RELATION2** by **removing all attributes not mentioned** in the operand (from both the schema and tuples) and **eliminating duplicate tuples**, keeping only unique ones.
In other words, projection allows selecting specific columns from a table.

Projection can be represented using the following notations (textual and graphical):

- **T = π {A1, A2, ..., Ap} (R)**
- **T = PROJECT (R / {A1, A2, ..., Ap})**
- **T = PROJECTION (R1, {A1, A2, ..., Ap})**



Example:

To display only the **last names and first names** of employees, we use one of the following expressions:

R1= π {NomEmp, PrenomEmp} (Employe)

R1= PROJECT (Employe / { NomEmp, PrenomEmp })

R1 = PROJECTION (Employe, { NomEmp, PrenomEmp})

Cette requête donne comme résultat la relation R1 suivante :

LastName	FirstName
Khaldi	Samir
Haroun	Ali
Heddad	Kawla

This query results in relation **R1**, which contains only the **LastName** and **FirstName** attributes.

3. SET OPERATORS

In relational algebra, **set operations** are fundamental for manipulating relations in the context of relational databases. They are often used to **combine, filter, or extract data** from multiple tables.

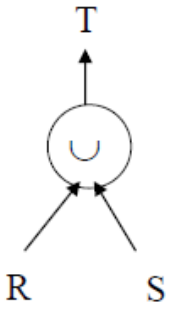
3.1. Union (∪)

Union is a fundamental operation in relational algebra, borrowed from set theory and adapted to relations with the **same schema**.
This operation is applied to two relations of the **same schema**, **RELATION1** and **RELATION2**, to construct a new relation **RELATION3**, which retains the same schema.

The resulting relation **RELATION3** contains all tuples belonging to **RELATION1, RELATION2, or both**.
The resulting relation maintains the attributes of the original relations and includes all tuples, **removing duplicates** if necessary.

Union can be represented using the following notations (textual and graphical):

- **T = R ∪ S**
- **T = UNION (R, S)**
- **T = APPEND (R, S)**



Example: Consider the two relations:

R1: Research Faculty Members

ID	FirstName	LastName
1	Falli	Ahmed
2	Deriche	Nacer
3	Salem	Halima
4	Batouche	Nacira

R2: Administrative Faculty Members

ID	FirstName	LastName
1	Falli	Ahmed
3	Salem	Halima
5	Mansour	Ridha
6	Amiche	Loubna

The complete list of faculty members is obtained with the query: **R1 ∪ R2**, producing the following result:

ID	FirstName	LastName
1	Falli	Ahmed
2	Deriche	Nacer
3	Salem	Halima
4	Batouche	Nacira
5	Mansour	Ridha
6	Amiche	Loubna

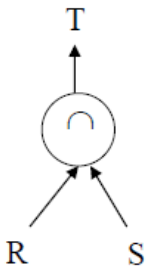
3.2. Intersection (∩)

Intersection is a fundamental operation in **relational algebra**, derived from **set theory** and adapted to relations with the **same schema**. This operation applies to two relations, **RELATION1** and **RELATION2**, to construct a new relation, **RELATION3**, which shares the same schema.

The resulting relation **RELATION3** contains only the **tuples that exist in both RELATION1 and RELATION2**.

Several notations have been introduced for this operation depending on the authors:

- **T = RELATION1 ∩ RELATION2**
- **T = INTERSECT (RELATION1, RELATION2)**
- **T = AND (RELATION1, RELATION2)**



Example

Using the two relations from the previous example, **R1 ∩ R2** gives the following relation:

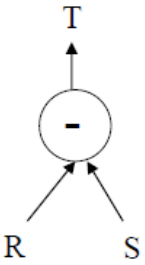
ID	FirstName	LastName
1	Falli	Ahmed
3	Salem	Halima

3.3. Difference

The **difference** is another fundamental operation from set theory, adapted for relations with the same schema. It applies to two relations, **RELATION1** and **RELATION2**, to construct a new relation, **RELATION3**, which contains tuples that belong to **RELATION1** but not to **RELATION2**.

The **difference** is a **non-commutative** operator, meaning the order of the operand relations is important. Several notations have been introduced for this operation, including:

- **T = RELATION1 – RELATION2**
- **T = DIFFERENCE (RELATION1, RELATION2)**
- **T = REMOVE (RELATION1, RELATION2)**
- **T = MINUS (RELATION1, RELATION2)**



The graphical notation shown above is also commonly used.

Example

Using the two relations from the previous example, **R1 - R2** gives the following relation:

ID	FirstName	LastName
2	Deriche	Nacer
4	Batouche	Nacira

On the other hand, **R2 – R1** gives the following relation:

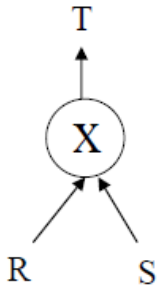
ID	FirstName	LastName
5	Mansour	Ridha
6	Amiche	Loubna

3.4. Cartesian Product

The Cartesian product is an operation applied to two relations, **RELATION1** and **RELATION2**, to construct a new relation, **RELATION3**. The schema of **RELATION3** is the concatenation of the schemas of the operand relations, and its tuples consist of all possible combinations of tuples from the operand relations.

Possible notations for this operation include:

- **T = RELATION1 × RELATION2**
- **T = PRODUCT (RELATION1, RELATION2)**
- **T = TIMES (RELATION1, RELATION2)**



The graphical notation shown above is also commonly used.

Example

IdA	*NameA*
1	*X*
2	*Y*

IdB	*NameB*
1	*G*
2	*T*

The cartesian product of these two relations gives the fellowing relation :

IdA	*NameA*	*IdB*	*NameB*
1	*X*	*1*	*G*
1	*X*	*2*	*T*
2	*Y*	*1*	*G*
2	*Y*	*2*	*T*

3.5. Division

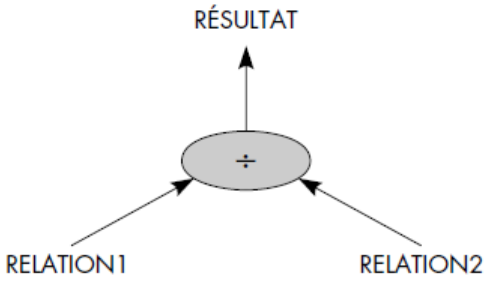
The division of a relation **R(X, Y)** by a relation **S(Y)** is the projection of **R** on **X**, restricted to tuples that are associated with all tuples of **S**.

Formal definition:

$R(X,Y) \div S(Y) = T(X) = \{ <x> \mid \forall\ y, <y> \in S \Rightarrow <x,y> \in R \}$

Possible notations for division:

- **D ÷ d**
- **DIVISION (D, d)**



Example:
Given the following two relations:

Employee

ID_Emp	ID_Prj
1	101
1	102
1	103
2	101
2	102
3	103

Projet

ID_Prj
101
102
103

The algebraic expression to find the employees who work on all projects is as follows:

Employee ÷ Projet

This query returns the following result:

ID_Emp
1

3.6 The Join

A join operation consists of associating tuples from two relations, **RELATION1** and **RELATION2**, based on a specified condition to form a third relation, **RELATION3**. This resulting relation contains all tuples obtained by concatenating a tuple from **RELATION1** and a tuple from **RELATION2** that satisfy the join condition.

The join of two relations thus produces a third relation that includes all combinations of tuples from the initial relations that meet the specified condition. The condition must ensure a meaningful association between the two relations and must follow the format:

<Attribute1> <operator> <Attribute2>

where **Attribute1** belongs to **RELATION1** and **Attribute2** belongs to **RELATION2**. Depending on the type of operator, we distinguish:

- **Equi-Join**: When the operator is =, which represents a true relational composition in the mathematical sense.
- **Non-Equi Join**: When the operator is one of the following: $<, \leq, \geq, >, \neq$.

In the case of **Equi-Join**, both attributes involved in the equality condition appear separately in the result, leading to duplication of values within each tuple. To eliminate this redundancy, the **Natural Join** is defined as follows:

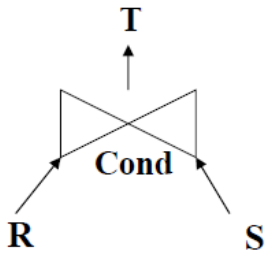
3.6.1 Natural Join (Jointure Naturelle)

A natural join operation associates tuples from two relations, **RELATION1** and **RELATION2**, to form a third relation, **RELATION3**. The attributes of **RELATION3** are the union of the attributes from **RELATION1** and **RELATION2**, and its tuples are obtained by combining a tuple from **RELATION1** and a tuple from **RELATION2** that share the same values for attributes with identical names.

- The join operation applies to two relations that must have at least one attribute defined in the same domain (i.e., the set of permissible values for an attribute).
- The join condition can be based on the equality of one or more attributes defined within the same domain (though they do not necessarily need to have the same name).
- The tuples in the resulting relation are formed by concatenating tuples from the original relations that satisfy the join condition.

The join operation is represented using one of the following notations. In the case of a **Natural Join**, the condition is omitted, implying that attributes with the same name are automatically matched based on equality.

- $\text{RELATION1} \bowtie_{\textit{Condition}} \text{RELATION2}$
- JOIN (RELATION1, RELATION2, Condition)



3.6.2 Outer Join (Jointure externe)

- An **outer join** returns all tuples from one or both relations, depending on the specified join condition.

- Tuples that do not find a match in the other relation will have **NULL** values for that relation's attributes.
- There are three types of outer joins: **Left Outer Join, Right Outer Join, and Full Outer Join**.

Left Outer Join (Jointure Externe Gauche)

Operator: R ⟕ S
Condition:

Returns all tuples from the **left relation R** and the matching tuples from the **right relation S**. If no tuple from **R** has a match, the attributes of **S** in the result will be **NULL**.

Example:
If you have an **"Employees"** table and a **"Departments"** table, a **left outer join** will return all employees, including those who do not belong to any department, with **NULL** values for the department attributes.

Right Outer Join (Jointure Externe Droite)

Operator: R ⟖ S
Condition:

Similar to the **left outer join**, but includes all tuples from **S** and only the matching tuples from **R**. If no tuple from **S** has a match, the attributes of **R** in the result will be **NULL**.

Example:
A **right outer join** between the **"Employees"** and **"Departments"** tables will return all **departments**, including those without any employees, with **NULL** values for the employee attributes.

Full Outer Join (Jointure Externe Complète)

Operator: R ⟗ S
Condition:

Returns all tuples from both **R** and **S**. If a tuple has no match in the other relation, the missing attributes will be **NULL**.

Example:
A **full outer join** between **"Employees"** and **"Departments"** will return **all employees and all departments**, with **NULL** values in cases where there is no corresponding match.

Semi-Joins and Anti-Joins

Left Semi Join (Demi-Jointure Gauche)

Operator: R ⋉ S

Returns all tuples from the **first table (R)** for which there exists a match in the **second table (S)**. The columns from **S** are **not included** in the result.

Example:
If you have an **"Employees"** table and a **"Departments"** table, the **left semi join** (Employees ⋉ Departments) will return **all employees who belong to a department**, but without the department attributes.

Right Semi Join (Demi-Jointure Droite)

Operator: R ⋊ S

Returns all tuples from the **second table (S)** for which there exists a match in the **first table (R)**. The columns from **R** are **not included** in the result.

Example:
A **right semi join** (Employees ⋊ Departments) will return **all departments that have at least one employee**, without including employee attributes.

Anti Join (Anti-Jointure)

Operator: R ▷ S

An **anti-join** is similar to a join, but it **only returns tuples from the first relation (R) that have no match in the second relation (S)**.

Example:
An **anti-join** (Employees ▷ Departments) will return **all employees who do not belong to any department**.

Some Properties of Join Operators

(You can now list specific properties related to joins, such as associativity, commutativity for certain types, and performance considerations in SQL.)

- $\sigma_{c1 \wedge c2}(R) = \sigma_{c1}(\sigma_{c2}(R))$
- $\sigma_{c1}(\sigma_{c2}(R)) = \sigma_{c2}(\sigma_{c1}(R))$
- $\Pi_{liste1}(\Pi_{liste2}(R)) = \Pi_{liste1}(R)$
- $R \bowtie_{C} S = S \bowtie_{C} R$
- $R \bowtie_{C1} (S \bowtie_{C2} T) = (S \bowtie_{C1} R) \bowtie_{C2} T$

4. ALGEBRAIC TREE

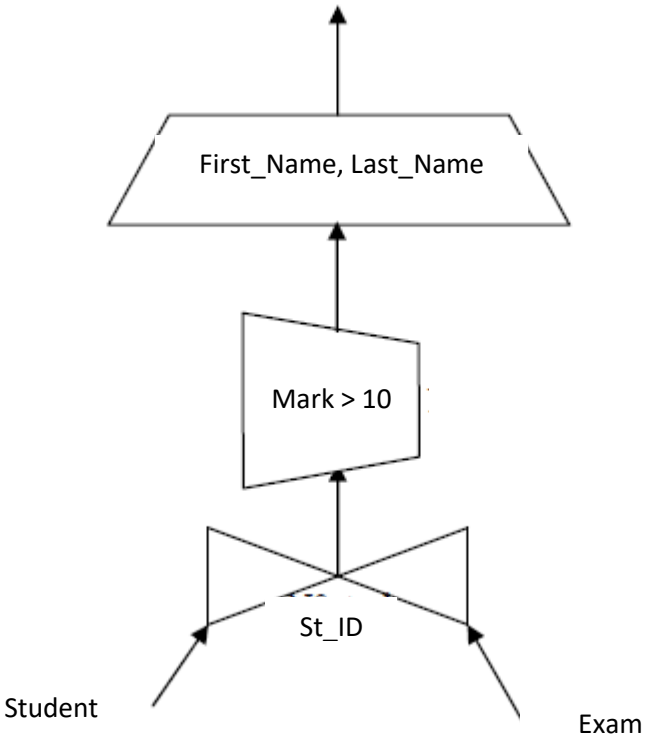
An algebraic tree corresponds to a decomposition of the query into elementary operators with the introduction of intermediate tables. It is therefore equivalent to rewriting the query using views. For the same question, multiple SQL queries can be equivalent.

Several trees can represent the same query, depending on the order in which operations are performed. A simple method to generate a tree consists of taking the qualification predicates in the order in which they appear, associating them with the corresponding relational operation, and then ending with a final projection to obtain the result.

For example, consider a database composed of the following relations:

- **Student** (St_ID, First_name, Last_name)
- **Module** (Module_ID, title)
- **Exam** (St_ID, Module_ID, grade)

To answer the following question: *" First_name and Last_name of all students who obtained a grade higher than 10,"* we can design the following operation tree:


First_Name, Last_Name
Mark > 10
St_ID
Student
Exam