📊 Module 3: Algebra & Querying

What Happens When You Ask a Database a Question?

🌟 Lesson Objective

By the end of this module, you will:

- Understand the basic operations of **relational algebra** and how they relate to table logic
- Learn how relational algebra maps to simple **SQL queries**
- Use **small example tables** to visualize set-based operations
- See why query order and structure can affect performance

🧠 Why This Matters

When you ask a question using SQL, the database doesn't just "look up" the answer—it transforms and filters sets of rows based on underlying rules. Relational algebra is the theoretical language that explains *how* that process works.

You won't be writing algebra expressions in real life, but understanding them will help you:

- Write better queries
- Understand what SQL is doing behind the scenes
- Improve performance by thinking like the database engine

📊 Section 1: The Building Blocks of Relational Algebra

📘 **Relational Algebra** is a formal system that describes how to manipulate sets of rows from tables. Each operation takes one or more tables (called **relations**) and produces a new table as a result.

Let's start with the most basic operations.

🔁 1. Selection (Filtering Rows)

Think of this as filtering specific rows from a table based on a condition.

Table: Players

PlayerID	Name	Age	MembershipStatus
101	Sam	32	Active
102	Jordan	29	Inactive
103	Taylor	41	Active

Goal: Get all active players

Result:

PlayerID	Name	Age	MembershipStatus
101	Sam	32	Active
103	Taylor	41	Active

SQL Equivalent:

```
SELECT * FROM Players WHERE MembershipStatus = 'Active';
```

📢 2. Projection (Selecting Columns)

This operation lets you focus on specific columns from a table.

Goal: Show only player names and ages

Result:

Name	Age
Sam	32
Jordan	29
Taylor	41

SQL Equivalent:

```
SELECT Name, Age FROM Players;
```

🟢 3. Union (Merging Unique Rows from Two Tables)

Combines all rows from two tables and removes duplicates.

ClubA_Players:

Name
Sam
Taylor

ClubB_Players:

Name
Jordan
Taylor

Result:

Name
Sam
Taylor
Jordan

SQL Equivalent:

```
SELECT * FROM ClubA_Players
UNION
SELECT * FROM ClubB_Players;
```

🔴 4. Set Difference (What's in One Table but Not the Other)

Returns rows in the first table that are not present in the second.

Registered_2024:

Name
Sam
Taylor

Registered_2025:

Name
Taylor

Result:

Name
Sam

SQL Equivalent:

```
SELECT * FROM Registered_2024
EXCEPT
SELECT * FROM Registered_2025;
```

🔵 5. Intersection (Shared Rows)

Returns only the rows that appear in both tables.

ClubA_Players:

Name
Sam
Taylor

ClubB_Players:

Name
Jordan
Taylor

Result:

Name
Taylor

SQL Equivalent:

```
SELECT * FROM ClubA_Players
INTERSECT
SELECT * FROM ClubB_Players;
```

🔗 6. Cartesian Product (All Combinations of Rows)

Combines every row from one table with every row from another.

Players:

PlayerID	Name
101	Sam
102	Jordan

Matches:

MatchID	Court
201	CourtA
202	CourtB

Result:

PlayerID	Name	MatchID	Court
101	Sam	201	CourtA
101	Sam	202	CourtB
102	Jordan	201	CourtA
102	Jordan	202	CourtB

SQL Equivalent:

```
SELECT * FROM Players, Matches;
```

🔀 Section 2: Introducing Joins

📘 A **join** connects related rows from two tables—usually using a **primary key** and a **foreign key**.

Let's define both terms before proceeding:

- **Primary Key**: A column (or set of columns) that uniquely identifies each row in a table.
- **Foreign Key**: A column in one table that refers to the primary key in another table, creating a relationship between the two.

Joins are built on a **Cartesian product** plus a filter condition.

Let's say we want player names and match dates:

Players Table:

PlayerID	Name
101	Sam
102	Jordan

Matches Table:

MatchID	Player1ID	Date
201	101	2024-06-10
202	102	2024-06-11

Here:

- PlayerID in **Players** is a **primary key**
- Player1ID in **Matches** is a **foreign key** referring to PlayerID

Goal: Match player names to their match dates

Result:

Name	Date
Sam	2024-06-10
Jordan	2024-06-11

SQL Equivalent:

```
SELECT Name, Date
FROM Players JOIN Matches
ON Players.PlayerID = Matches.Player1ID;
```

🚦 Section 3: Why Query Order and Structure Matter

Even though the result of a query is the same, the **order** of operations can affect performance:

Example:

```
-- Slow version
SELECT * FROM Players JOIN Matches ON ... WHERE Age > 40;

-- Faster version
SELECT * FROM (SELECT * FROM Players WHERE Age > 40) AS p
JOIN Matches ON p.PlayerID = Matches.Player1ID;
```

Filtering **early** reduces how much data is joined.

📘 *This is called query optimization. You don't need to master it now—but keep it in mind as queries get more complex.*

✅ Self-Check

1. What does filtering rows look like in SQL?
2. What does the union operation do?
3. What kind of operation combines every row from two tables?
4. Why might filtering first be faster than joining first?
5. What is the difference between a primary key and a foreign key?

📘 Glossary

Term	Definition
Relational Algebra	A system of operations used to manipulate relational tables
Selection	Filters rows based on a condition
Projection	Selects specific columns
Cartesian Product	Combines every row from two tables
Join	Combines rows from two tables based on related keys
Union	Combines results from two tables, removing duplicates
Intersection	Returns only rows that appear in both tables
Difference	Returns rows that appear in one table but not the other
Primary Key	A column that uniquely identifies each row in a table
Foreign Key	A column that refers to a primary key in another table