Module 5: Structured Query Language (SQL) - Part 2

Welcome back! In Module 4, you learned the absolute basics of SQL: how to build the structure of your database (like drawing the blueprint for a house with `CREATE TABLE`) and how to put basic things in and out of those structures (`INSERT`, `SELECT` with `WHERE`, `UPDATE`, `DELETE`). That was like learning the alphabet and basic sentences in SQL.

Now, in Module 5, we're going to learn how to write much more powerful "sentences" in SQL. This is where SQL truly shines for making sense of large amounts of data. Imagine you have thousands of student records. You probably don't want to see every single one. You might want to know:

- "What's the *average* grade in this class?"
- "How many students are there *in each major*?"
- "Which departments have *a lot* of students?"
- "Show me all the students, *sorted* by their last name."
- "Combine information from the `Students` table and the `Departments` table so I can see each student's name *and* their department's name."
- "Find students who earn *more than the average salary* for their major."

To answer questions like these, we need **Advanced SQL Queries**. We'll learn how to summarize data, group it into categories, sort it, combine it from different tables, and even use the results of one query to help answer another. This module will give you the tools to become a true data explorer!

Chapter 5: Advanced SQL Queries

5.1 Aggregate Functions

Imagine you have a big pile of numbers, like all the test scores from a class. Instead of looking at each score, you might want to find out some quick facts about the whole class. **Aggregate functions** are like special calculators in SQL that take a *whole bunch of values* from many rows and squish them down into *one single result*. They give you a summary.

When you use an aggregate function by itself (without something called `GROUP BY`, which we'll learn soon), it will calculate its result using *all the rows* that your `FROM` and `WHERE` clauses found.

Let's look at the most common ones:

- **`COUNT()` - How Many?**
    - **What it does:** This function counts things. It tells you "how many" of something there are.
    - **Different ways to use it:**

        - `COUNT(*)`: This is like asking, "How many rows are there in total?" It counts *every single row*, even if some columns in those rows are empty (NULL).

Example: If you have a `Students` table, and you want to know the total number of students:

```
SQL
SELECT COUNT(*)
FROM Students;
```

            - *If your `Students` table has 100 rows, this will give you `100`.*
        - `COUNT(column_name)`: This is like asking, "How many rows have a *non-empty value* in this specific column?" It only counts rows where the `column_name` has an actual value. If it's `NULL` (meaning unknown or empty), that row is *not* counted for this specific column.

Example: If some students haven't provided an email address (their `Email` column is `NULL`), and you only want to count students with an email:

```
SQL
SELECT COUNT(Email)
FROM Students;
```

            - *If 95 students have an email and 5 don't, this will give you `95`.*
        - `COUNT(DISTINCT column_name)`: This is like asking, "How many *unique* non-empty values are there in this specific column?" It counts each different value only once.

Example: If your `Students` table has students from `MajorDeptID`s 10, 20, 10, 30, 20, 10, and you want to know how many *different* departments students are majoring in:

```
SQL
SELECT COUNT(DISTINCT MajorDeptID)
FROM Students;
```

            - *This will give you `3` (for DeptID 10, 20, and 30, assuming no NULLs).*
- **`SUM()` - What's the Total?**
    - **What it does:** Adds up all the numbers in a specified column.
    - **Important:** This only works for columns that contain numbers (like `INTEGER`, `DECIMAL`, `FLOAT`).
    - **Syntax:** `SUM(numeric_column_name)`

Example: To find the total amount of money paid in salaries to all employees:

```
SQL
SELECT SUM(Salary)
FROM Employees;
```

    - *If salaries are 50000, 60000, 70000, this will give you `180000`.*

- **`AVG()` - What's the Average?**

    - **What it does:** Calculates the average value (the sum of all numbers divided by the count of those numbers) in a specified column.
    - **Important:** This also only works for columns that contain numbers.
    - **Syntax:** `AVG(numeric_column_name)`

Example: To find the average grade point average (GPA) across all students:

```
SQL
SELECT AVG(GPA)
FROM Students;
```

    - *If GPAs are 3.0, 3.5, 4.0, this will give you `3.5`.*

- **`MIN()` - What's the Smallest?**

    - **What it does:** Finds the smallest value in a specified column.
    - **Works for:** Numbers, text (alphabetical order), and dates/times.
    - **Syntax:** `MIN(column_name)`

Example: To find the earliest birth date among all students (which helps you find the oldest student):

```
SQL
SELECT MIN(DateOfBirth)
FROM Students;
```

    - *If dates are 2004-01-01, 2003-05-10, 2005-09-15, this will give you `2003-05-10`.*

- **`MAX()` - What's the Largest?**

    - **What it does:** Finds the largest value in a specified column.
    - **Works for:** Numbers, text (alphabetical order), and dates/times.
    - **Syntax:** `MAX(column_name)`

Example: To find the highest salary paid to any employee:

```
SQL
SELECT MAX(Salary)
FROM Employees;
```

    - *If salaries are 50000, 60000, 70000, this will give you `70000`.*

Key takeaway for Aggregate Functions: They always take many rows as input and give you just one summary row as output, unless you use `GROUP BY`.

5.2 `GROUP BY` Clause

Now, let's connect the dots. You know how to get the `AVG()` salary for *all* employees. But what if you want the `AVG()` salary *for each department*? Or the `COUNT()` of students *in each major*? This is where the **`GROUP BY` clause** becomes incredibly useful.

The `GROUP BY` clause works hand-in-hand with aggregate functions. It tells SQL to divide your rows into smaller "groups" based on one or more columns you specify. Then, the aggregate functions you use in your `SELECT` statement will calculate their summaries *for each of these groups individually*.

Imagine this: You have a big stack of papers, and each paper is a student record.

- If you just use `COUNT(*)`, you count all papers in the stack (total students).
- If you want to `COUNT(*)` per major, you first `GROUP BY MajorDeptID`. This means you physically separate the papers into smaller stacks: one stack for "Computer Science majors," one for "Physics majors," etc. Then, you `COUNT(*)` the papers in *each individual stack*.

General Syntax:

SQL
SELECT column_to_group_by, another_column_to_group_by,
aggregate_function(some_column)
FROM table_name
WHERE condition_to_filter_rows -- (Optional: filters rows BEFORE grouping)
GROUP BY column_to_group_by, another_column_to_group_by;

Important Rules for `GROUP BY`:

1. **Select List Rule:** Any column that you put in your `SELECT` statement that is *not* inside an aggregate function (`COUNT`, `SUM`, `AVG`, `MIN`, `MAX`) *must* also be listed in your `GROUP BY` clause.
    - **Why?** Because if you group by `DeptID`, SQL can show you `DeptID` (since there's only one per group). But if you also try to select `EmployeeName` *without* an aggregate, which `EmployeeName` should it show? There could be many employees in one `DeptID` group. So, you can only select the columns you're grouping by, or columns inside aggregate functions.
2. **Order of Execution (Logical Flow):** It's helpful to understand the logical steps SQL takes:

- **1. `FROM`:** SQL first looks at the table(s) you specified.
- **2. `WHERE`:** If you have a `WHERE` clause, SQL filters out individual rows that don't meet the condition. Only the remaining rows continue.
- **3. `GROUP BY`:** SQL then takes these filtered rows and groups them based on the `GROUP BY` columns.
- **4. Aggregate Functions:** For each of these newly formed groups, SQL calculates the aggregate functions (e.g., `SUM`, `AVG`).
- **5. `SELECT`:** Finally, SQL selects the specified columns (the grouping columns and the aggregate results) for each group.

Examples:

Count the number of students in each major department:

```
 SQL
SELECT MajorDeptID, COUNT(StudentID) AS NumberOfStudents -- NumberOfStudents is an alias for readability
FROM Students
GROUP BY MajorDeptID;
```

-

Possible Result:

```
 MajorDeptID | NumberOfStudents
------------|-----------------
10          | 150
20          | 120
30          | 80
NULL        | 5    -- Students without a major
```

-

Find the average salary for each job title:

```
 SQL
SELECT JobTitle, AVG(Salary) AS AverageSalary
FROM Employees
GROUP BY JobTitle;
```

-

Find the earliest and latest enrollment dates for each major:

```
 SQL
SELECT MajorDeptID, MIN(EnrollmentDate) AS FirstEnrollment, MAX(EnrollmentDate) AS LastEnrollment
FROM Students
GROUP BY MajorDeptID;
```

-

5.3 `HAVING` Clause

You know that the `WHERE` clause filters individual rows *before* grouping. But what if you want to filter the *groups themselves*? For example, "Show me only departments that have *more than 100 students*," or "Which product categories have an *average price above $50*?"

You can't use `WHERE` for this because `WHERE` operates on individual rows *before* aggregates are calculated. That's where the **`HAVING` clause** comes in.

The `HAVING` clause is specifically designed to filter groups based on conditions that often involve aggregate functions. It's like a `WHERE` clause, but it applies to the results of the `GROUP BY` operation.

General Syntax:

SQL
```
SELECT column_to_group_by, aggregate_function(some_column)
FROM table_name
WHERE condition_on_rows     -- Optional: Filters individual rows BEFORE grouping
GROUP BY column_to_group_by
HAVING condition_on_groups; -- Filters the groups AFTER grouping and aggregation
```

Logical Order of Execution (Revisited with `HAVING`):

1. **`FROM`:** Identify tables.
2. **`WHERE`:** Filter individual rows (non-aggregated data).
3. **`GROUP BY`:** Group the remaining rows.
4. **Aggregate Functions:** Calculate summaries for each group.
5. **`HAVING`:** Filter the groups based on conditions, often using the results of the aggregate functions.
6. **`SELECT`:** Display the final results.

Examples:

Find major departments that have more than 100 students:

SQL
```
SELECT MajorDeptID, COUNT(StudentID) AS NumberOfStudents
FROM Students
GROUP BY MajorDeptID
HAVING COUNT(StudentID) > 100;
```

-
    - *Explanation:*

1. All students are grouped by `MajorDeptID`.
2. `COUNT(StudentID)` is calculated for each `MajorDeptID` group.
3. The `HAVING` clause then checks `COUNT(StudentID) > 100` for *each group's count*. Only groups meeting this condition are shown.

List departments where the average employee salary is greater than $60,000:

SQL
```
SELECT DeptID, AVG(Salary) AS AverageSalary
FROM Employees
GROUP BY DeptID
HAVING AVG(Salary) > 60000;
```

-

Find product categories where the minimum price is at least $20, but only for products that are currently in stock (Quantity > 0):

SQL
```
SELECT Category, MIN(Price) AS MinPrice, COUNT(*) AS NumProducts
FROM Products
WHERE Quantity > 0 -- Filters individual products
GROUP BY Category
HAVING MIN(Price) >= 20 AND COUNT(*) > 5; -- Filters groups
```

-
    - *Note the combined use of `WHERE` and `HAVING`: `WHERE` removes out-of-stock items first, then `GROUP BY` groups the remaining items, and `HAVING` filters those groups based on `MIN(Price)` and `COUNT(*)`.*

5.4 `ORDER BY` Clause

The `ORDER BY` clause is all about presentation. After SQL has figured out which rows to select, grouped them (if applicable), and filtered them (if applicable), the `ORDER BY` clause tells SQL how you want the final results to be sorted when they are displayed to you. It does *not* change the way data is stored in the database; it only affects the output of your query.

Imagine this: You've made your list of students. Now you want to sort that list alphabetically by last name, or by GPA from highest to lowest.

General Syntax:

SQL
```
SELECT column1, column2, ...
FROM table_name
[WHERE condition]
[GROUP BY columns]
[HAVING condition]
```

ORDER BY column_or_expression1 [ASC | DESC], column_or_expression2 [ASC | DESC], ...;

- `ORDER BY`: The keywords that indicate you want to sort the results.
- `column_or_expression`: You specify the column(s) by which you want to sort. You can also sort by an expression, like an aggregate function result (e.g., `ORDER BY COUNT(StudentID)`).
- `[ASC | DESC]`: These are optional keywords:
    - `ASC`: Stands for **Ascending**. This sorts from smallest to largest (e.g., A-Z for text, 0-9 for numbers, earliest to latest date). This is the **default** behavior if you don't specify `ASC` or `DESC`.
    - `DESC`: Stands for **Descending**. This sorts from largest to smallest (e.g., Z-A for text, 9-0 for numbers, latest to earliest date).

Sorting by Multiple Columns (Primary and Secondary Sorts): You can list multiple columns in your `ORDER BY` clause, separated by commas. SQL will sort by the first column listed. If there are rows that have the same value in the first column (a "tie"), then SQL will use the second column to sort those tied rows. If there are still ties, it moves to the third column, and so on.

Logical Position: The `ORDER BY` clause is always the **very last** logical operation in a `SELECT` statement. It operates on the final result set produced by all the other clauses.

Examples:

Sort all students by their `LastName` alphabetically (ascending):

SQL
```
SELECT StudentID, FirstName, LastName
FROM Students
ORDER BY LastName ASC; -- 'ASC' is optional here as it's the default
```

-

Sort employees by `Salary` from highest to lowest, and if salaries are the same, then by `FirstName` alphabetically:

SQL
```
SELECT EmpID, EmpName, Salary
FROM Employees
ORDER BY Salary DESC, FirstName ASC;
```

-
    - *Result:* Employees with higher salaries appear first. If two employees have the same salary, the one whose first name comes earlier alphabetically will appear first among those with that salary.

List departments, sorted by the number of students they have, from most students to fewest:

SQL

```
SELECT MajorDeptID, COUNT(StudentID) AS NumberOfStudents
FROM Students
GROUP BY MajorDeptID
ORDER BY NumberOfStudents DESC; -- Sorting by the aggregated result
```

-

5.5 SQL Joins

In Module 2, we learned about the **Relational Model** and how data is often split across multiple tables to avoid repetition (data redundancy) and ensure accuracy (data integrity) using Primary Key-Foreign Key relationships. For example, student details are in a `Students` table, and department details are in a `Departments` table, linked by `MajorDeptID` (FK) and `DeptID` (PK).

When you need to get information that combines data from these different, but related, tables, you use **Joins**. A `JOIN` operation combines rows from two or more tables based on a related column between them. It's how you bring scattered pieces of related information together into one meaningful result.

Let's imagine our `Students` table and `Departments` table:

Students Table:

StudentID	FirstName	LastName	MajorDeptID
101	Alice	Smith	10
102	Bob	Johnson	20
103	Carol	Davis	10
104	David	Lee	NULL

Export to Sheets

Departments Table:

DeptID	DeptName	Location
10	Computer Science	Main Campus
20	Physics	North Campus
30	Chemistry	East Campus

Export to Sheets

5.5.1 `INNER JOIN` (or just `JOIN`)

- **What it does:** This is the most common and default type of join. An `INNER JOIN` creates a new result table by combining rows from two tables ONLY when there is a **match** in the specified common column(s) in *both* tables. If a row in one table doesn't have a matching row in the other table, it is *not* included in the result.

- **Analogy:** Imagine you have a list of students and a list of departments. An `INNER JOIN` is like finding all the students who *actually have* a matching department listed in the department table. Students without a major (`NULL`) or with a `MajorDeptID` that doesn't exist in the `Departments` table will be left out.

Syntax:

```
 SQL
SELECT columns_you_want
FROM table1
INNER JOIN table2
ON table1.common_column = table2.common_column;
```

-
    - `ON`: This keyword specifies the **join condition**. It's usually where you state how the two tables are linked (e.g., `Students.MajorDeptID = Departments.DeptID`).
    - **Table Aliases:** It's good practice to use short aliases (like `S` for `Students`, `D` for `Departments`) for table names, especially when dealing with multiple tables or self-joins. This makes the query shorter and clearer.

Example: Get the `FirstName`, `LastName` of students and the `DeptName` of their major department.

```
 SQL
SELECT S.FirstName, S.LastName, D.DeptName
FROM Students AS S
INNER JOIN Departments AS D
ON S.MajorDeptID = D.DeptID;
```

-

Result based on example tables:

```
 FirstName | LastName | DeptName
----------|----------|-----------------
Alice     | Smith    | Computer Science
Bob       | Johnson  | Physics
```

Carol | Davis | Computer Science

-
- **Explanation:** `David Lee` (StudentID 104) is *not* in the result because his `MajorDeptID` is `NULL`, which doesn't match any `DeptID` in the `Departments` table. Also, `Chemistry` (DeptID 30) is *not* in the result because no student has `MajorDeptID = 30`.

5.5.2 `LEFT JOIN` (or `LEFT OUTER JOIN`)

- **What it does:** A `LEFT JOIN` returns *all rows* from the **left table** (the first table mentioned in your `FROM` clause) and the matching rows from the **right table**. If there's no match for a row in the left table in the right table, the columns from the right table will show `NULL` values for that row.
- **Analogy:** "Show me *all* the students, and if they have a major, show me its name. If they don't have a major, still show the student, but put `NULL` for the department name."

Syntax:
SQL
```
SELECT columns_you_want
FROM table1  -- This is the LEFT table
LEFT JOIN table2 -- This is the RIGHT table
ON table1.common_column = table2.common_column;
```

-

Example: Get all student names and their major department names (if any).
SQL
```
SELECT S.FirstName, S.LastName, D.DeptName
FROM Students AS S -- Students is the LEFT table
LEFT JOIN Departments AS D -- Departments is the RIGHT table
ON S.MajorDeptID = D.DeptID;
```

-

Result based on example tables:
```
 FirstName | LastName | DeptName
-----------|----------|-----------------
Alice      | Smith    | Computer Science
Bob        | Johnson  | Physics
Carol      | Davis    | Computer Science
David      | Lee      | NULL            -- David's MajorDeptID is NULL, so no match from Departments
```

-
- **Explanation:** Notice that `David Lee` is now included, but his `DeptName` is `NULL` because there was no matching department for his `MajorDeptID`

(which was `NULL`). The `Chemistry` department (DeptID 30) is still *not* in the result because it was only asked to keep all rows from the *left* table (`Students`).

5.5.3 `RIGHT JOIN` (or `RIGHT OUTER JOIN`)

- **What it does:** A `RIGHT JOIN` is the opposite of a `LEFT JOIN`. It returns *all rows* from the **right table** (the second table mentioned in your `FROM` clause) and the matching rows from the **left table**. If there's no match for a row in the right table in the left table, the columns from the left table will show `NULL` values.
- **Analogy:** "Show me *all* the departments, and if they have students, show me their names. If a department has no students, still show the department, but put `NULL` for the student name."

Syntax:

```
 SQL
SELECT columns_you_want
FROM table1 -- This is the LEFT table
RIGHT JOIN table2 -- This is the RIGHT table
ON table1.common_column = table2.common_column;
```

-

Example: Get all department names and any students who major in them.

```
 SQL
SELECT S.FirstName, S.LastName, D.DeptName
FROM Students AS S -- Students is the LEFT table
RIGHT JOIN Departments AS D -- Departments is the RIGHT table
ON S.MajorDeptID = D.DeptID;
```

-

Result based on example tables:

```
 FirstName | LastName | DeptName
-----------|----------|-----------------
Alice      | Smith    | Computer Science
Bob        | Johnson  | Physics
Carol      | Davis    | Computer Science
NULL       | NULL     | Chemistry       -- Chemistry (DeptID 30) has no matching students
```

  -
  - **Explanation:** Here, `Chemistry` (DeptID 30) is included even though no student majors in it. The student columns (`FirstName`, `LastName`) are `NULL` for this row. `David Lee` (who had `MajorDeptID = NULL`) is *not* in this result because he did not match any `DeptID` from the `Departments` table.

- **Note:** In practice, `RIGHT JOIN` can almost always be rewritten as a `LEFT JOIN` by simply swapping the order of the tables in the `FROM` clause. Most developers stick to `LEFT JOIN` for consistency.

5.5.4 `FULL JOIN` (or `FULL OUTER JOIN`)

- **What it does:** A `FULL JOIN` returns all rows when there is a match in *either* the left table or the right table. It's essentially a combination of a `LEFT JOIN` and a `RIGHT JOIN`. If a row from one table has no match in the other, the corresponding columns from the non-matching table will show `NULL` values.
- **Analogy:** "Show me *all* students and *all* departments. Match them up where possible. If a student has no major, show them with a `NULL` department. If a department has no students, show it with `NULL` student info."

Syntax:
SQL
```
SELECT columns_you_want
FROM table1
FULL JOIN table2
ON table1.common_column = table2.common_column;
```

-
- *Note:* Not all database systems fully support `FULL JOIN` (e.g., MySQL does not have a direct `FULL JOIN` keyword and requires a combination of `LEFT JOIN`, `RIGHT JOIN`, and `UNION`).

Example: Get all students and all departments, regardless of whether they have a match.
SQL
```
SELECT S.FirstName, S.LastName, D.DeptName
FROM Students AS S
FULL JOIN Departments AS D
ON S.MajorDeptID = D.DeptID;
```

-

Result based on example tables (assuming DBMS supports FULL JOIN):
```
 FirstName | LastName | DeptName
-----------|----------|-----------------
Alice      | Smith    | Computer Science
Bob        | Johnson  | Physics
Carol      | Davis    | Computer Science
David      | Lee      | NULL            -- Student with no matching DeptID
NULL       | NULL     | Chemistry       -- Department with no matching student
```

    -

    - **Explanation:** This query provides a complete picture, showing students who have majors, students who don't, and departments that have students, and departments that don't.

5.5.5 `SELF-JOIN`

- **What it does:** Sometimes, the relationship you want to explore exists within a *single* table. A `SELF-JOIN` is when you join a table with itself. This might seem strange, but it's very useful for finding relationships between different rows of the same table. A common example is an `Employees` table where an `EmployeeID` is linked to a `ManagerID` (which is also an `EmployeeID`).

- **Key Technique:** To perform a `SELF-JOIN`, you *must* use **table aliases**. You treat the same table as if it were two separate, distinct tables during the query.

Example: Assume an `Employees` table: **Employees Table:** | EmpID | EmpName | ManagerID | | :---- | :---------- | :-------- | | 1 | Alice | NULL | | 2 | Bob | 1 | | 3 | Carol | 1 | | 4 | David | 2 |

We want to list each employee and their manager's name.

SQL
SELECT E.EmpName AS EmployeeName, M.EmpName AS ManagerName
FROM Employees AS E -- Treat this as the "employee" instance of the table
INNER JOIN Employees AS M -- Treat this as the "manager" instance of the table
ON E.ManagerID = M.EmpID; -- Join where the employee's ManagerID matches the manager's EmpID

-

Result:
EmployeeName	ManagerName
Bob	Alice
Carol	Alice
David	Bob

    -
    - **Explanation:** We effectively created two "copies" of the `Employees` table (aliased as `E` and `M`) and joined them to find the employee-manager relationships. Alice is not listed as an `EmployeeName` because she has no `ManagerID`.

5.5.6 `CROSS JOIN`

- **What it does:** A `CROSS JOIN` creates the **Cartesian Product** of two tables. This means it combines *every single row* from the first table with *every single row* from the second table. No join condition (`ON` clause) is specified for a `CROSS JOIN`.
- **Analogy:** If you have 3 shirts and 4 pairs of pants, a `CROSS JOIN` would give you all 12 possible shirt-and-pants combinations.

Syntax:
SQL
```
SELECT columns_you_want
FROM table1
CROSS JOIN table2;
```

-
    - **Caution:** `CROSS JOIN` can produce extremely large result sets (`number_of_rows_in_table1 * number_of_rows_in_table2`). Use it with extreme caution and only when you truly need every possible combination.

Example: (Using our `Students` and `Departments` tables)
SQL
```
SELECT S.FirstName, D.DeptName
FROM Students AS S
CROSS JOIN Departments AS D;
```

-

Result based on example tables: (4 students * 3 departments = 12 rows)
```
 FirstName | DeptName
----------|-----------------
Alice     | Computer Science
Alice     | Physics
Alice     | Chemistry
Bob       | Computer Science
Bob       | Physics
Bob       | Chemistry
... (and so on for Carol and David, paired with all 3 departments)
```

    -
    - **Use Cases:** `CROSS JOIN` is rarely used directly for common data retrieval. Its main uses are in specific scenarios like generating test data, creating combinations, or as a fundamental building block for more complex join types (though modern SQL typically handles this implicitly).

5.6 Subqueries (Nested Queries)

Sometimes, to answer a question, you need to first ask *another* question to the database. This is where **subqueries** (also called **nested queries** or **inner queries**) come in. A subquery is simply a `SELECT` statement written inside another SQL query.

The subquery always runs *first*, and its result is then used by the outer (main) query. Subqueries are always enclosed in **parentheses `()`**.

Think of it: "First, find X. Then, use X to find Y."

Rules for Subqueries:

- They must be enclosed in parentheses `()`.
- They run before the outer query.
- They can return a single value, a single row, or a whole table.
- They can be used in `SELECT`, `FROM`, `WHERE`, and `HAVING` clauses.

Let's look at the different types based on what they return:

5.6.1 Scalar Subqueries

- **What it returns:** A single value (one row, one column).
- **Where you can use it:** Anywhere a single value or an expression is expected, such as in the `SELECT` list, `WHERE` clause, `HAVING` clause, or `SET` clause of an `UPDATE` statement.

Example: Find the names of employees whose salary is *greater than the overall average salary* of all employees.
SQL
SELECT EmpName, Salary
FROM Employees
WHERE Salary > (SELECT AVG(Salary) FROM Employees);

-
    - **Explanation:**
        1. The inner subquery `(SELECT AVG(Salary) FROM Employees)` runs first. It calculates the average salary for *all* employees (e.g., `65000`).
        2. The outer query then becomes `SELECT EmpName, Salary FROM Employees WHERE Salary > 65000;`. It filters the employees based on this single value.

5.6.2 Row Subqueries

- **What it returns:** A single row, which can contain one or more columns.
- **Where you can use it:** Typically in the `WHERE` or `HAVING` clause, usually for comparisons where you need to match multiple column values at once (row-wise comparison).

Example: Find the `StudentID` of any student who has the *exact same first name and last name* as StudentID 101.
SQL
SELECT StudentID, FirstName, LastName

```
FROM Students
WHERE (FirstName, LastName) = (SELECT FirstName, LastName FROM Students
WHERE StudentID = 101);
```

-
    - **Explanation:**
        1. The inner subquery `(SELECT FirstName, LastName FROM Students WHERE StudentID = 101)` returns a single row like `('Alice', 'Smith')`.
        2. The outer query then looks for rows where *both* `FirstName` is 'Alice' *and* `LastName` is 'Smith'.

5.6.3 Table Subqueries (Derived Tables / Inline Views)

- **What it returns:** A complete table, with one or more rows and one or more columns.
- **Where you can use it:**
    - In the `FROM` clause, where it acts as a temporary table that the outer query can select from (often called a **derived table** or **inline view**). When used in the `FROM` clause, it *must* be given an alias.
    - With operators like `IN`, `EXISTS`, `ANY`, `ALL` (covered in the next section).

Example (Derived Table): Find the average salary for departments that have more than 5 employees.

```
 SQL
SELECT DeptStats.DeptID, DeptStats.AverageSalary
FROM (
    SELECT DeptID, AVG(Salary) AS AverageSalary, COUNT(EmpID) AS NumEmployees
    FROM Employees
    GROUP BY DeptID
    HAVING COUNT(EmpID) > 5
) AS DeptStats; -- The subquery result is a temporary table named DeptStats
```

-
    - **Explanation:**
        1. The inner subquery first calculates the average salary and employee count for each department, but only keeps departments with more than 5 employees.
        2. This result becomes a temporary table called `DeptStats`.
        3. The outer query then simply selects the `DeptID` and `AverageSalary` from this temporary `DeptStats` table. This approach makes complex queries more modular and readable.

5.7 `ANY`, `ALL`, `EXISTS`, `IN` Operators with Subqueries

These operators are powerful tools used with subqueries in the `WHERE` or `HAVING` clauses to create very specific conditions. They help you compare a value to a *set* of values returned by a subquery.

- **`IN` Operator:**
    - **What it does:** Checks if a value is *equal to any* value in the list (or set) returned by the subquery. It's like writing many `OR` conditions.
    - **Syntax:** `WHERE column_name IN (subquery_that_returns_a_single_column_list)`

Example: Find the names of students who are majoring in 'Computer Science' or 'Physics'.
SQL
```
SELECT FirstName, LastName
FROM Students
WHERE MajorDeptID IN (SELECT DeptID FROM Departments WHERE DeptName IN ('Computer Science', 'Physics'));
```

    -
        - **Explanation:**
            - The subquery `(SELECT DeptID FROM Departments WHERE DeptName IN ('Computer Science', 'Physics'))` returns a list of department IDs, say `(10, 20)`.
            - The outer query then becomes `WHERE MajorDeptID IN (10, 20)`, filtering students whose `MajorDeptID` is either 10 or 20.
- **`EXISTS` Operator:**
    - **What it does:** Checks for the *existence* of any rows returned by a subquery. It returns `TRUE` if the subquery returns at least one row, and `FALSE` otherwise. The specific values returned by the subquery don't matter, only whether *any* row is returned. `EXISTS` is very efficient because the subquery can stop executing as soon as it finds the first matching row. It's often used with **correlated subqueries** (where the inner query depends on values from the outer query).
    - **Syntax:** `WHERE EXISTS (subquery)`

Example: Find departments that have at least one employee.
SQL
```
SELECT DeptName
FROM Departments D
WHERE EXISTS (SELECT 1 FROM Employees E WHERE E.DeptID = D.DeptID);
```

    -
        - **Explanation:** For each department `D` from the `Departments` table:
            - The subquery `(SELECT 1 FROM Employees E WHERE E.DeptID = D.DeptID)` is run. It tries to find *any* employee (`E`) whose `DeptID` matches the *current* `DeptID` from the outer `Departments` table (`D.DeptID`).

        - If the subquery finds even one such employee, `EXISTS` returns `TRUE`, and that `DeptName` is included in the result. If no such employee is found, `EXISTS` returns `FALSE`.
- **`ANY` (or `SOME`) Operator:**
    - **What it does:** Compares a value to *any* value in the set returned by a subquery. The condition is `TRUE` if the comparison is true for *at least one* value in the subquery's result set. It's often used with comparison operators (`=`, `>`, `<`, etc.).
    - **Syntax:** `WHERE expression comparison_operator ANY (subquery_that_returns_a_single_column_list)`

Example: Find employees whose salary is greater than the salary of *any* employee in the 'Sales' department.

SQL

```
SELECT EmpName, Salary
FROM Employees
WHERE Salary > ANY (SELECT Salary FROM Employees WHERE DeptID = (SELECT DeptID FROM Departments WHERE DeptName = 'Sales'));
```

    - **Explanation:** Let's say 'Sales' department salaries are (50000, 60000, 70000).
        - `Salary > ANY (50000, 60000, 70000)` means: `Salary > 50000 OR Salary > 60000 OR Salary > 70000`.
        - So, if an employee has a salary of 55000, this condition is `TRUE` because 55000 is greater than 50000.
- **`ALL` Operator:**
    - **What it does:** Compares a value to *every single* value in the set returned by a subquery. The condition is `TRUE` only if the comparison is true for *all* values returned by the subquery.
    - **Syntax:** `WHERE expression comparison_operator ALL (subquery_that_returns_a_single_column_list)`

Example: Find employees whose salary is greater than the salary of *all* employees in the 'Sales' department.

SQL

```
SELECT EmpName, Salary
FROM Employees
WHERE Salary > ALL (SELECT Salary FROM Employees WHERE DeptID = (SELECT DeptID FROM Departments WHERE DeptName = 'Sales'));
```

    - **Explanation:** Using the same 'Sales' salaries (50000, 60000, 70000):

        - `Salary > ALL (50000, 60000, 70000)` means: `Salary > 50000 AND Salary > 60000 AND Salary > 70000`.
        - For this to be `TRUE`, an employee's salary would need to be greater than 70000.

5.8 Set Operations

Set operations allow you to combine the results of two or more independent `SELECT` statements into a single result set. They treat the output of each `SELECT` statement as a "set" of rows and then perform standard set theory operations on them.

Crucial Requirement: Union Compatibility For set operations to work, the `SELECT` statements being combined must be **union compatible**. This means they must:

1. Return the **same number of columns**.
2. Have **corresponding columns with compatible data types** in the same order (e.g., if the first column of the first query is a number, the first column of the second query must also be a number).

Let's assume we have a `Students` table and a `Faculty` table with some common columns (like `FirstName`, `LastName`, `City`).

- **`UNION`:**
    - **What it does:** Combines the result sets of two or more `SELECT` statements and automatically **removes any duplicate rows** from the final result.

Syntax:
SQL
```
SELECT column1, column2 FROM table1
UNION
SELECT column_a, column_b FROM table2;
```
    -

Example: Get a unique list of all first names and last names of everyone in the university (both students and faculty).
SQL
```
SELECT FirstName, LastName FROM Students
UNION
SELECT FirstName, LastName FROM Faculty;
```
    -
        - *Explanation:* If 'John Smith' is listed as both a student and a faculty member, he will only appear once in the final combined list.
- **`UNION ALL`:**

    - **What it does:** Combines the result sets of two or more `SELECT` statements and **keeps all duplicate rows**. It does not perform the extra step of removing duplicates, making it generally faster than `UNION`.

Syntax:
```
SQL
SELECT column1, column2 FROM table1
UNION ALL
SELECT column_a, column_b FROM table2;
```

    -

Example: Get a full list of all first names and last names of everyone (students and faculty), even if there are duplicates.
```
SQL
SELECT FirstName, LastName FROM Students
UNION ALL
SELECT FirstName, LastName FROM Faculty;
```

    -
        - *Explanation:* If 'John Smith' is both a student and a faculty member, he will appear twice in the final combined list.
- **`INTERSECT`:**

    - **What it does:** Returns only the rows that are **common to *both* `SELECT` statements' result sets**. In other words, it finds the overlap.

Syntax:
```
SQL
SELECT column1, column2 FROM table1
INTERSECT
SELECT column_a, column_b FROM table2;
```

    -
    - *Note:* The `INTERSECT` operator is supported in databases like PostgreSQL, Oracle, and SQL Server. However, **MySQL does not have a direct `INTERSECT` keyword**. In MySQL, you would achieve the same result using `INNER JOIN` or `EXISTS` with a subquery.

Example: Find individuals who are listed as *both* a student *and* a faculty member (based on their first and last name).
```
SQL
SELECT FirstName, LastName FROM Students
INTERSECT
SELECT FirstName, LastName FROM Faculty;
```

    -

- **`EXCEPT` (or `MINUS` in Oracle):**
    - **What it does:** Returns the rows that are present in the **first `SELECT` statement's result set but are *not* present in the second `SELECT` statement's result set.** It's like finding what's unique to the first set when compared to the second.

Syntax:
SQL
```
SELECT column1, column2 FROM table1
EXCEPT
SELECT column_a, column_b FROM table2;
```

    -
    - *Note:* The `EXCEPT` operator is supported in databases like PostgreSQL and SQL Server. In Oracle, it's called `MINUS`. **MySQL does not have a direct `EXCEPT` or `MINUS` keyword**. In MySQL, you would typically achieve this result using `LEFT JOIN` with `WHERE ... IS NULL` or `NOT EXISTS`.

Example: Find students who are *not* also listed as faculty members.
SQL
```
SELECT FirstName, LastName FROM Students
EXCEPT
SELECT FirstName, LastName FROM Faculty;
```

    -

Module Summary

You've just completed a deep dive into the truly powerful aspects of SQL queries! This module has equipped you with the advanced tools needed to perform sophisticated data analysis and retrieval, moving far beyond simple data selection.

We started by understanding **Aggregate Functions** (`COUNT`, `SUM`, `AVG`, `MIN`, `MAX`), which allow you to quickly summarize large datasets into single, actionable values. To apply these summaries to specific categories of your data, you mastered the **`GROUP BY` clause**, learning how to group rows with common characteristics. To filter these summarized groups based on aggregate results, you learned the crucial role of the **`HAVING` clause**, distinguishing its purpose from the row-level filtering of `WHERE`. To ensure your query results are always presented in a clear and organized manner, you gained expertise in the **`ORDER BY` clause** for sorting your output.

A cornerstone of relational databases is the ability to connect information across different tables, and you thoroughly explored this with **SQL Joins**. You now understand the nuanced differences and applications of `INNER JOIN` (for exact matches), `LEFT JOIN` (to include all

from the left), `RIGHT JOIN` (to include all from the right), and `FULL JOIN` (to include all from both). You also learned how to use a **`SELF-JOIN`** to find relationships within a single table and understood the concept of a `CROSS JOIN`.

Finally, you advanced your query construction skills with **Subqueries (Nested Queries)**, learning how to embed one `SELECT` statement within another to solve complex problems. You differentiated between **Scalar**, **Row**, and **Table** subqueries based on their return type and purpose. Complementing subqueries, you gained proficiency in the powerful conditional operators: `ANY`, `ALL`, `EXISTS`, and `IN`, which allow for highly specific and dynamic filtering based on subquery results. You concluded by understanding **Set Operations** (`UNION`, `UNION ALL`, `INTERSECT`, `EXCEPT`), enabling you to combine or compare the results of multiple independent queries.

With the detailed knowledge and practical examples from this module, you are now well-equipped to design and execute complex SQL queries, unlocking deep insights from your relational databases and preparing you for even more advanced database system