Subqueries, Views, CASE

Contents

[1. Subqueries 2](#_Toc187249332)

[Problem Statement: 1 2](#_Toc187249333)

[Problem Statement: 2 6](#_Toc187249334)

[Problem Statement: 3 8](#_Toc187249335)

[Problem Statement: 4 12](#_Toc187249336)

[Problem Statement: 5 15](#_Toc187249337)

[Problem Statement: 6 17](#_Toc187249338)

[Problem Statement: 7 18](#_Toc187249339)

[Problem Statement: 8 22](#_Toc187249340)

[2. Views 26](#_Toc187249341)

[Problem with FORM Clause Subqueries 26](#_Toc187249342)

[Problems with Creating Actual Big Fat Tables 26](#_Toc187249343)

[Introduction to Views 27](#_Toc187249344)

[Problem Statement 27](#_Toc187249345)

# Subqueries

### Definition of Subqueries:

* Subqueries are a way to solve complex problems in SQL by breaking them into smaller, more intuitive queries.
* Subqueries are not essential, as most problems can also be solved without them. However, they provide an easier or more intuitive way to write queries.

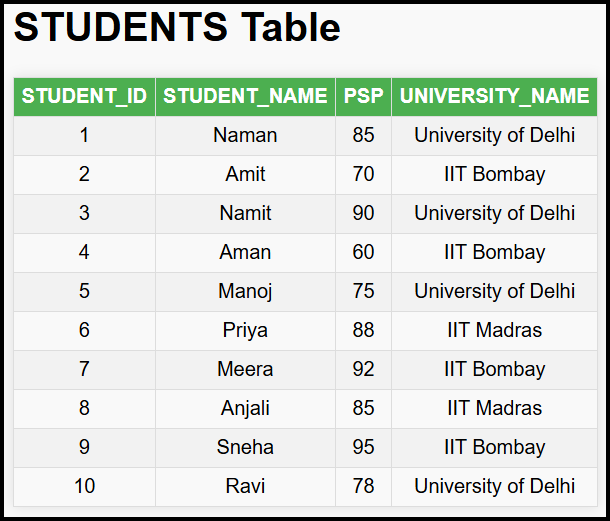
### Purpose of Subqueries:

* They help simplify complex queries by dividing them into manageable parts.
* Subqueries make complex problem statements more understandable and easier to implement.

### Key Point about Subqueries:

* Anything achievable with subqueries can likely be achieved without them, but subqueries make the query more readable and maintainable.

## Problem Statement: 1

* Given a table STUDENTS with columns: STUDENT\_ID, STUDENT\_NAME, PSP (Problem Solving Percentage), UNIVERSITY\_NAME, find all students whose PSP is greater than the PSP of the student with STUDENT\_ID = 2.
* Created a table using the below SQL.

CREATE TABLE STUDENTS (

    STUDENT\_ID INT AUTO\_INCREMENT PRIMARY KEY,

    STUDENT\_NAME VARCHAR (50),

    PSP INT,

    UNIVERSITY\_NAME VARCHAR (100)

);

* Inserted few rows…

INSERT INTO

    STUDENTS (STUDENT\_NAME, PSP, UNIVERSITY\_NAME)

VALUES

    ('Naman', 85, 'University of Delhi'),

    ('Amit', 70, 'IIT Bombay'),

    ('Namit', 90, 'University of Delhi'),

    ('Aman', 60, 'IIT Bombay'),

    ('Manoj', 75, 'University of Delhi'),

    ('Priya', 88, 'IIT Madras'),

    ('Meera', 92, 'IIT Bombay'),

    ('Anjali', 85, 'IIT Madras'),

    ('Sneha', 95, 'IIT Bombay'),

    ('Ravi', 78, 'University of Delhi');

* The above problem can be solved using joins…

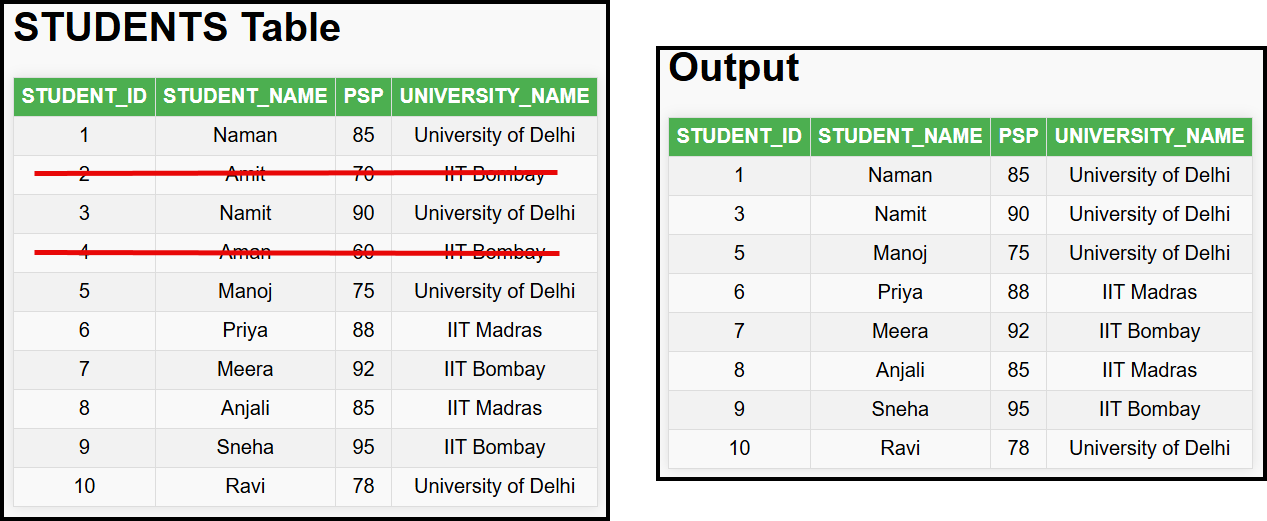
SELECT

    S1.\*

FROM

    STUDENTS S1

    JOIN STUDENTS S2 ON S2.STUDENT\_ID = 2

    AND S1.PSP > S2.PSP;

### Solution without Subqueries: Using Self-Joins

* Use a self-join to compare rows within the same table.
* Concept: JOIN the STUDENTS table with itself and filter based on the condition that the PSP of one student (S1) is greater than the PSP of another student (S2) with STUDENT\_ID = 2.

### Query Explanation

1. **Setup the Self-Join:**
   1. Create two aliases for the STUDENTS table: S1 and S2.
   2. Join these tables on the condition that the STUDENT\_ID of S2 is 2.
2. **Filter Rows:**
   1. Include rows where S1.PSP (PSP of the first student) is greater than S2.PSP (PSP of the student with STUDENT\_ID = 2).

### Programming Approach:

**Query Purpose:** The given SQL query finds students with a PSP greater than any PSP of student with STUDENT\_ID = 2.

1. In programming, solving this problem involves:
   * Finding PSP of student with a specific STUDENT\_ID (e.g., STUDENT\_ID = 2):
   * Example in pseudo-code:

for student in students:

    if student.id == 2:

        psp = student.psp

1. Comparing PSP values for all students:
   * Example in pseudo-code:

for student in students:

    if student.psp > psp:

        result\_list.append(student)

### **Using Subqueries in SQL:**

* **Step 1:** Find the PSP of a student with STUDENT\_ID 2:

SELECT

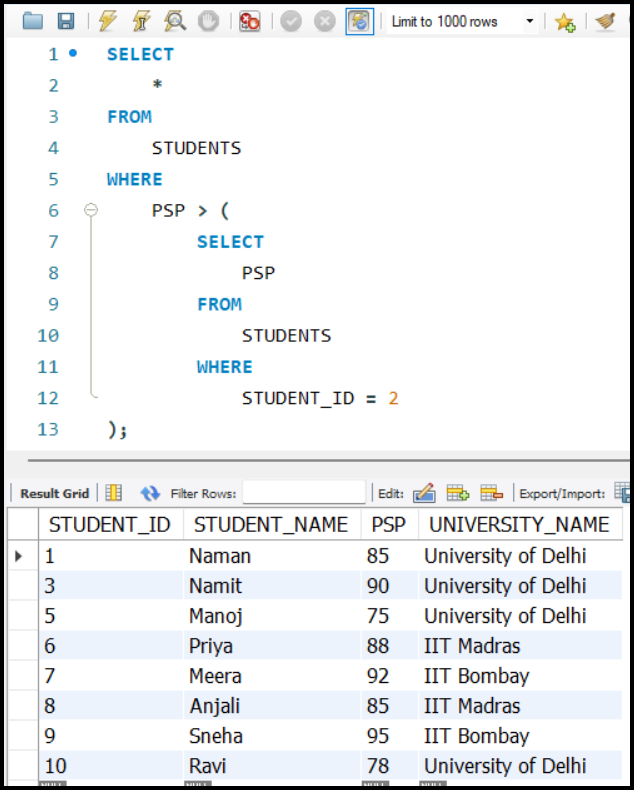
    PSP

FROM

    STUDENTS

WHERE

    STUDENT\_ID = 2;

* **Step 2:** Find students whose PSP is greater than the PSP from **Step 1**:

SELECT

    \*

FROM

    STUDENTS

WHERE

    PSP > (

        SELECT

            PSP

        FROM

            STUDENTS

        WHERE

            STUDENT\_ID = 2

);

### **Advantages of Subqueries:**

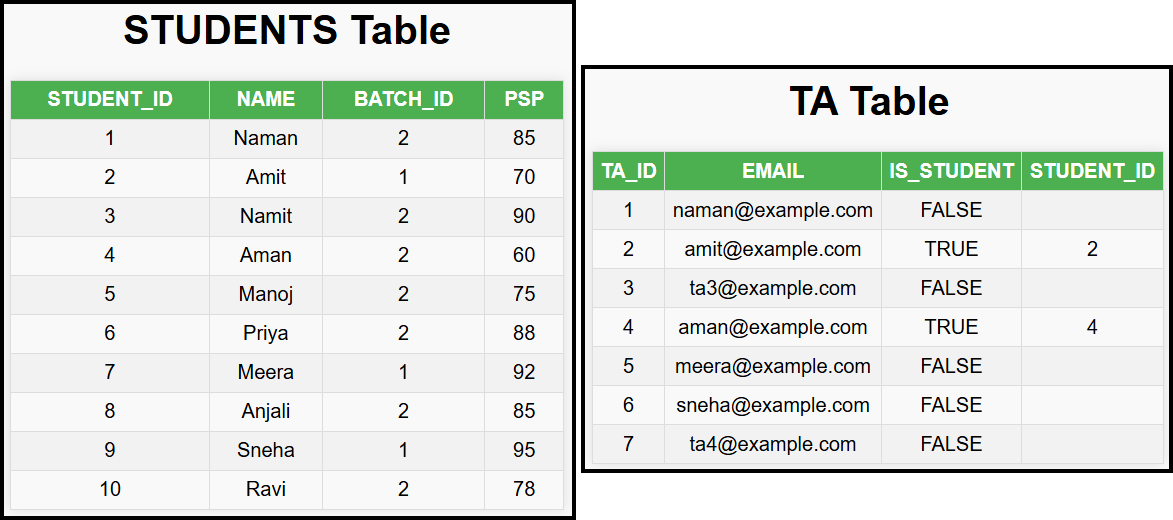
* **Intuitive and Readable:** The subquery approach closely resembles a programming approach and is easier to understand.

### **Performance Considerations:**

1. **Time Complexity of Subqueries:**
   * The subquery runs once for every row in the outer query.
   * If there are n rows in the table, the subquery will execute n times, resulting in O(n²) complexity.
2. **Optimizations:**
   * **SQL Query Optimizers:** Modern optimizers can avoid redundant subquery execution by caching results.
   * **Indexing:** Proper indexing on relevant columns significantly reduces query execution time.

## Problem Statement: 2

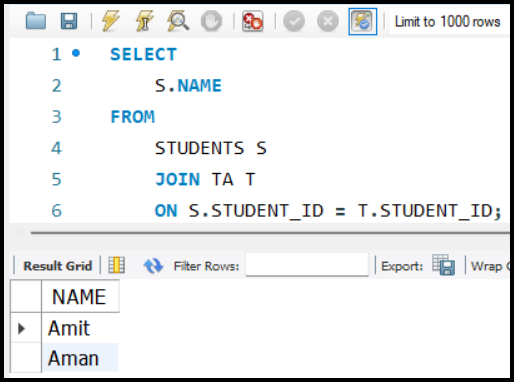
* Two tables are provided:
  + STUDENTS Table: Contains STUDENT\_ID, NAME, BATCH\_ID, and PSP.
  + TA Table: Contains TA\_ID, EMAIL, IS\_STUDENT (Boolean), and STUDENT\_ID.
* **Objective**: Print the names of STUDENTS who are also TAs.
  + IS\_STUDENT is TRUE if the person is both a student and a TA.
  + STUDENT\_ID is the ID of the student in the STUDENTS Table, or NULL if not a student.



### Approach

#### Join Approach:

* + Use an INNER JOIN between STUDENTS and TA tables.
  + Condition: STUDENTS.ID = TA.STUDENT\_ID.



SELECT

    S.NAME

FROM

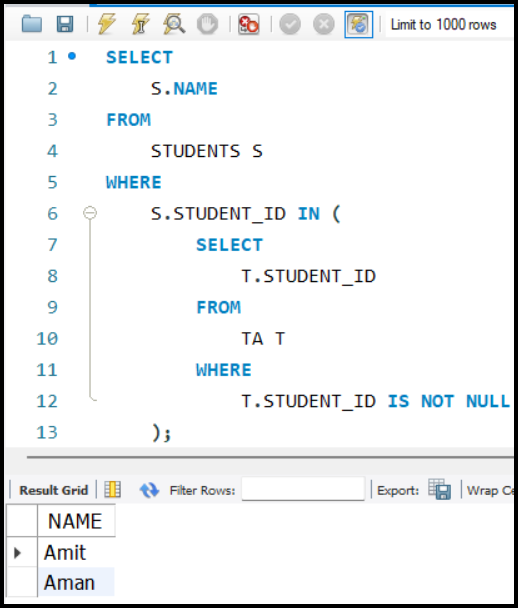
    STUDENTS S

    JOIN TA T ON S.STUDENT\_ID = T.STUDENT\_ID;

* This will return names of STUDENTS who are also TAs.
* The join ensures only matching rows (STUDENTS who are TAs) are included.

#### Alternative Subquery Approach:

* Use the IN keyword to filter STUDENTS whose IDs are present in the TA table:

SELECT

    S.NAME

FROM

    STUDENTS S

WHERE

    S.STUDENT\_ID IN (

        SELECT

            T.STUDENT\_ID

        FROM

            TA T

        WHERE

            T.STUDENT\_ID IS NOT NULL

    );

* The IN keyword allows a subquery to return a list of Student\_IDs from the TA table.
* The outer query checks if a student's ID exists in this list.

### Programming Approach:

* Brute Force Approach:
  + Iterate over both lists (Students and TA) to check matching conditions.
  + Pseudocode:

answer = []

for student in students:

    for ta in tas:

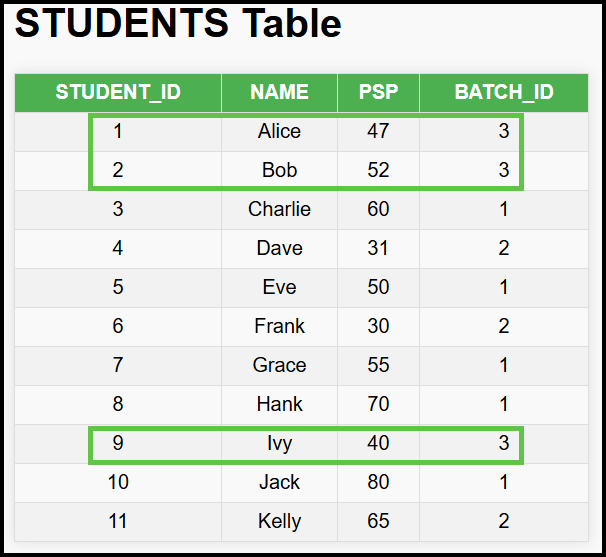
        if ta.Student\_ID == student.ID:

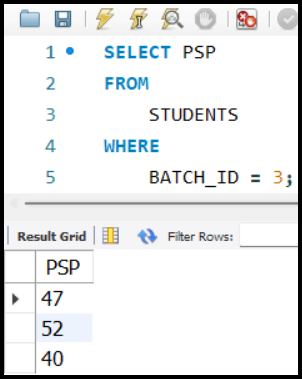
            answer.append(student.Name)

* + - For each student, iterate through all TAs.
    - If the Student\_ID of the TA matches the ID of the student, add the student’s name to the result.
* **Drawbacks**:
  + Time complexity: O(n×m), where n is the number of students and m is the number of TAs.

## Problem Statement: 3

* **We have** STUDENTS Table: Contains STUDENT\_ID, NAME, BATCH\_ID, and PSP.
* **Objective: Find** STUDENTS **with** PSP> **Every Student of** BATCH\_ID = 3.



* Task:
  + Identify students whose PSP is greater than the PSP of all students in BATCH\_ID 3.
* Steps:
  + Step 1: Find the PSPs of all students in BATCH\_ID 3:

SELECT

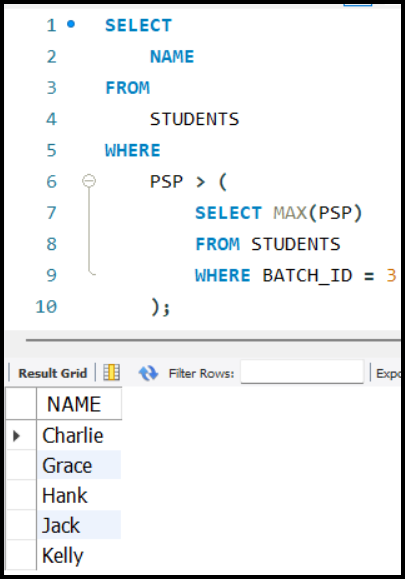
PSP

FROM

    STUDENTS

WHERE

    BATCH\_ID = 3;

* + Step 2: Check if a student's PSP is greater than all the values returned in Step1.

SELECT

    NAME

FROM

    STUDENTS

WHERE

    PSP > (

        SELECT MAX(PSP)

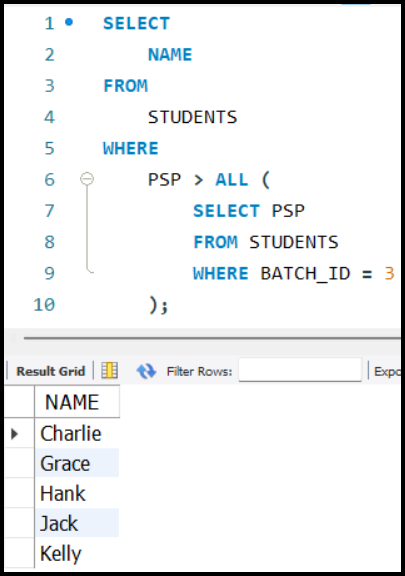
        FROM STUDENTS

        WHERE BATCH\_ID = 3

    );

* + - For each student in the main table, compare their PSP with the list of PSPs retrieved in Step1.

### ALL Clause

* Definition:
  + The ALL clause is used to compare a value against all rows returned by a subquery.
* How It Works:
  + If the condition is TRUE for all rows in the subquery's result, the record satisfies the condition.

SELECT

    NAME

FROM

    STUDENTS

WHERE

    PSP > ALL (

        SELECT PSP

        FROM STUDENTS

        WHERE BATCH\_ID = 3

    );

* Subquery: Retrieves the PSPs of all students in BATCH\_ID 3 which is [47, 52, 40].
* Condition: The PSP of each student in the main query is compared with all the PSPs returned by the subquery.
  + For each student in the main table:
    - Check if their PSP satisfies the condition (e.g., >).
    - The condition must hold true for **every row** returned by the subquery.
  + Example Analysis:
    - If PSP = 47:
      * Compare PSP > 47 → False.
      * Compare PSP > 52 → True.
      * Result: Not included in the final output.
    - If PSP = 60 (STUDENT NAME = 'Charlie'):
      * Compare PSP > 47 → True.
      * Compare PSP > 52 → True.
      * Result: Included in the final output.

### **Comparison Between IN and ALL**

* IN **Clause:**
  + Checks if a value is **present** in a list of values.
  + Example:

SELECT

    NAME

FROM

    STUDENTS

WHERE

    PSP IN (

        SELECT PSP

        FROM STUDENTS

        WHERE BATCH\_ID = 3

    );

* ALL **Clause:**
  + Checks if a condition is true for **all values** in the list.
  + Example:

SELECT

    NAME

FROM

    STUDENTS

WHERE

    PSP > ALL (

        SELECT PSP

        FROM STUDENTS

        WHERE BATCH\_ID = 3

    );

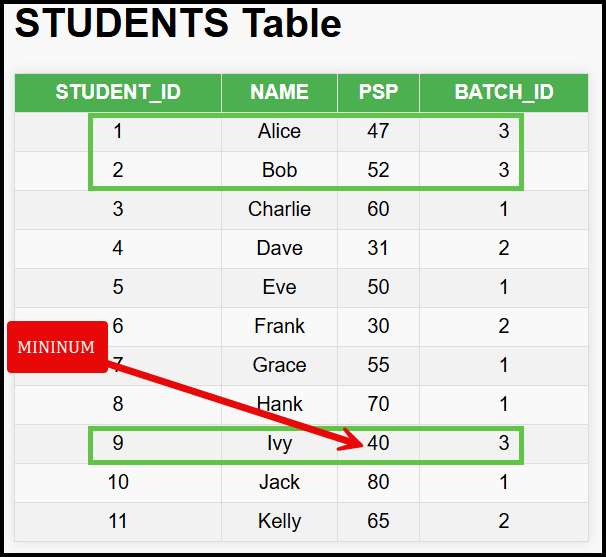
### Summary:

#### **Example of Using the ALL Clause**

* **Condition**: We want to find values that are greater than all the values in the list.
  + Example:
    - PSP = 80:
      * Is 80 greater than 41? Yes.
      * Is 80 greater than 32? Yes.
      * Is 80 greater than 47? Yes.
    - **Result**: 80 is part of the final result.
    - PSP = 46:
      * Is 46 greater than 41? Yes.
      * Is 46 greater than 32? Yes.
      * Is 46 greater than 47? No.
    - **Result**: 46 is not part of the final result.
* The ALL clause is used to compare a value against all values in a given list.
* **Syntax**: VALUE [COMPARISON OPERATOR] ALL (LIST OF VALUES).
  + Example: WHERE PSP > ALL (41, 32, 47).

## Problem Statement: 4

* **We have** STUDENTS Table: Contains STUDENT\_ID, NAME, BATCH\_ID, and PSP.
* Find all STUDENTS whose PSP (Problem Solving Percentage) is greater than any of the PSP values of students in BATCH\_ID 3. Even if one student's PSP in BATCH\_ID 3 is less than a given student's PSP, that student should be included in the result.



### Using MIN

* Query:

SELECT

    NAME, PSP

FROM

    STUDENTS

WHERE

    PSP > (

        SELECT MIN(PSP)

        FROM STUDENTS

        WHERE BATCH\_ID = 3

    );

* Explanation:
  + Subquery: Finds the minimum PSP of students in BATCH\_ID 3.
  + Condition: Selects STUDENTS whose PSP is greater than this minimum value.

### Using ANY Clause

* Query:

SELECT

    NAME, PSP

FROM

    STUDENTS

WHERE

    PSP > ANY (

        SELECT PSP

        FROM STUDENTS

        WHERE BATCH\_ID = 3

    );

* Explanation:
  + Subquery: Retrieves all PSP values of students in BATCH\_ID 3.
  + ANY Clause: Checks if a student’s PSP is greater than any PSP value in the list.

### ANY Clause Mechanics:

* The ANY clause evaluates whether a condition is true for any value in a list.
  + Example:

SELECT NAME

FROM STUDENTS

WHERE PSP > ANY (30, 41, 20);

* + - For PSP = 43:
      * 43 > 30 → True
      * Since one condition is true, 43 is included in the result.
    - For PSP = 24:
      * 24 > 30 → False
      * 24 > 41 → False
      * 24 > 20 → True
    - Since one condition is true, 24 is included.
    - For PSP = 18:
      * 18 > 30 → False
      * 18 > 41 → False
      * 18 > 20 → False
    - Since no condition is true, 18 is not included.

#### ANY vs ALL:

* ANY:
  + Equivalent to a logical OR operation.
  + At least one condition must be true for the row to be included.
* ALL:
  + Equivalent to a logical AND operation.
  + All conditions must be true for the row to be included.

#### Summary:

* **Syntax**: VALUE [COMPARISON OPERATOR] ANY (LIST OF VALUES).
  + Example: WHERE PSP > ANY (30, 41, 20).

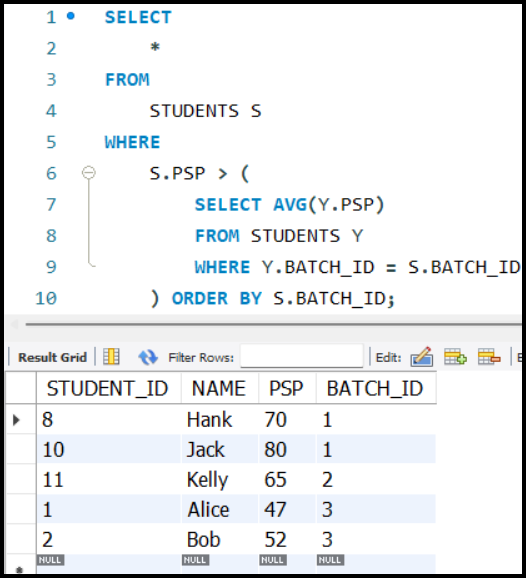
## Problem Statement: 5

* **We have** STUDENTS Table: Contains STUDENT\_ID, NAME, BATCH\_ID, and PSP.
* **Objective**: Find all STUDENTS whose PSP is greater than the average PSP of their batch.
  + Calculate average PSP for each batch and compare each student's PSP with the average PSP of their batch.

### **Algorithm in Programming Language**

1. Iterate through all rows of the STUDENTS table.
2. For each student, compute the average PSP of the batch they belong to.
3. Compare the PSP of the student with the computed average PSP.
4. Include the student in the result if their PSP is greater.

### SQL Query Breakdown

* Step 1: Select FROM Students (FROM is like iterating over each student).
  + Query: SELECT \* FROM STUDENTS
* Step 2: Apply Conditions Using WHERE Clause (WHERE is like ‘if’ condition).
  + Query: WHERE PSP > (SELECT AVG(PSP) FROM STUDENTS WHERE BATCH\_ID = STUDENTS.BATCH\_ID)

### SQL Query:

SELECT

    \*

FROM

    STUDENTS S

WHERE

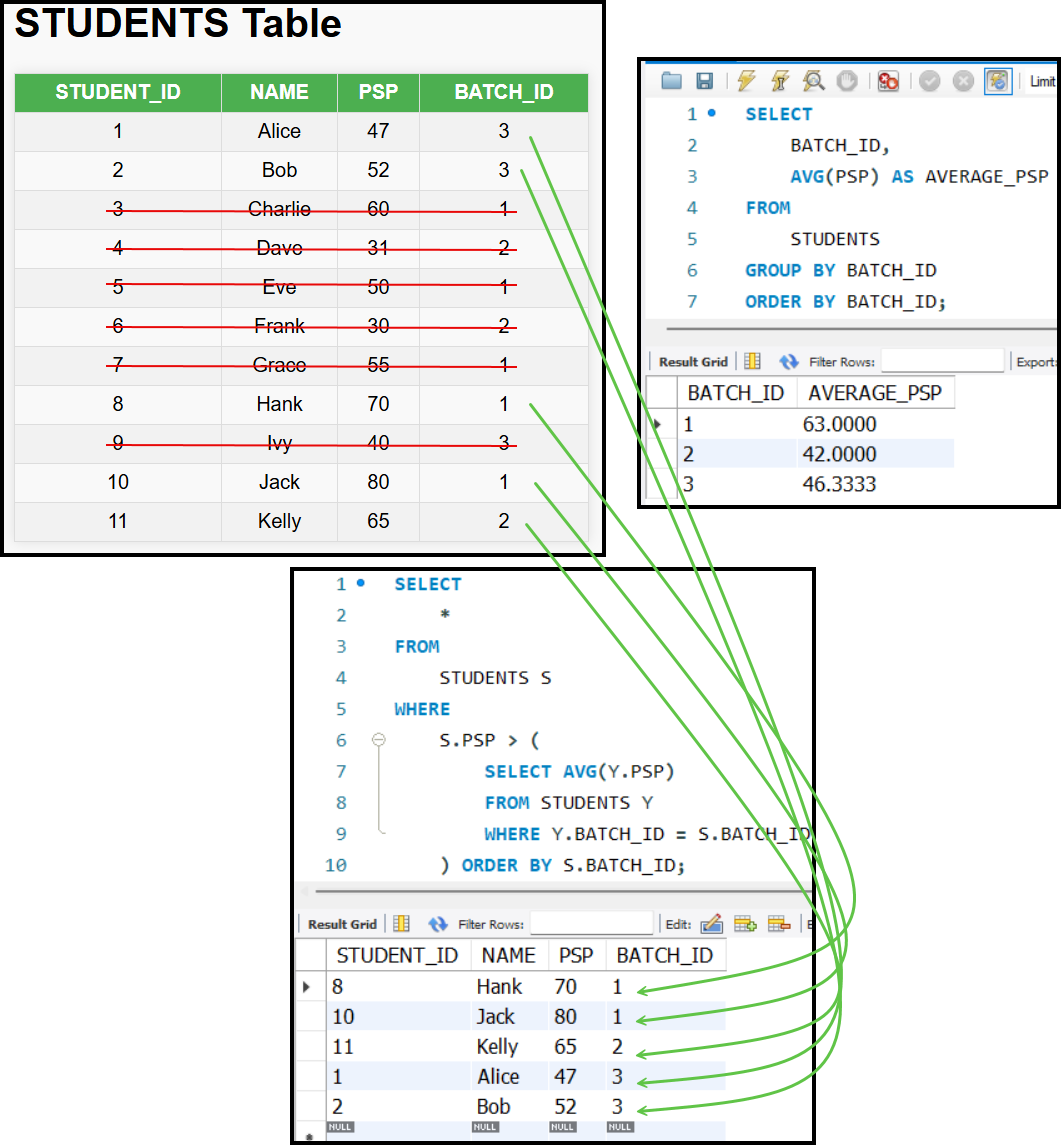
    S.PSP > (

        SELECT AVG(Y.PSP)

        FROM STUDENTS Y

        WHERE Y.BATCH\_ID = S.BATCH\_ID

    ) ORDER BY S.BATCH\_ID;



### Concepts:

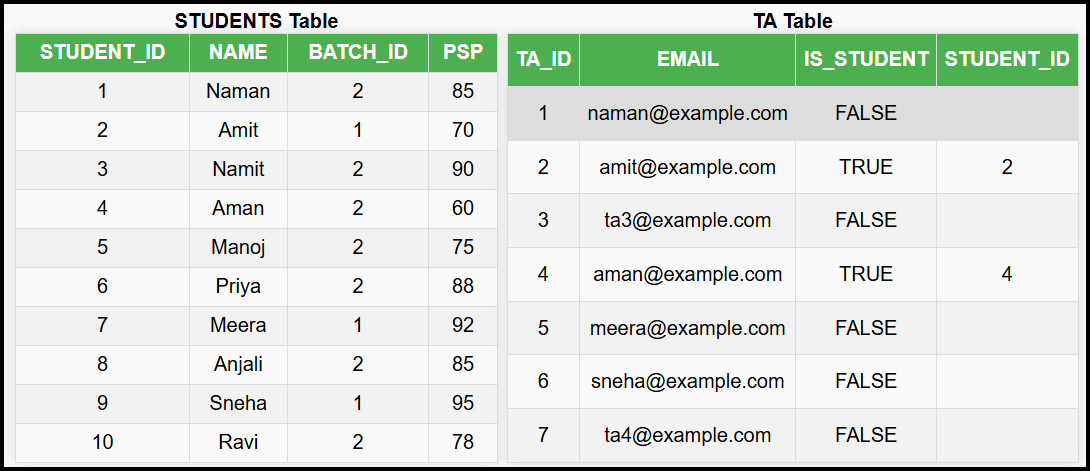
* Correlated Subqueries:
  + The subquery is dependent on the row from the main query.
  + Example: S.BATCH\_ID in the subquery relates to the current row in the main query.
* Subquery Behaviour:
  + Returns a different value for each row of the main query based on correlation.

### Correlated Subqueries

* Definition: A subquery that depends on the outer query for its values.
* Use Case: When the subquery needs to return different values for different rows of the outer query.
* Example:
  + Outer Query: SELECT \* FROM STUDENTS S
  + Correlated Subquery: SELECT AVG(PSP) FROM STUDENTS WHERE BATCH\_ID = S.BATCH\_ID

## Problem Statement: 6

* **Objective**: Find the names of STUDENTS who are also TAs.



### EXISTS Clause

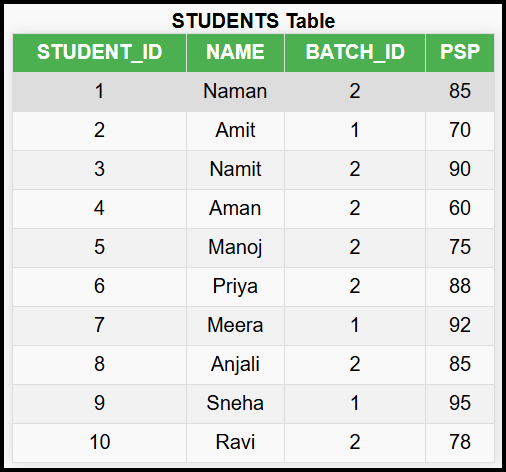
* Query

|  |
| --- |
| SELECT NAME  FROM STUDENTS S  WHERE      EXISTS (          SELECT \*          FROM TA          WHERE TA.STUDENT\_ID = S.STUDENT\_ID      ); |

* Explanation:
  + The outer query fetches NAME from the STUDENTS table.
  + The EXISTS clause checks for a condition:
    - If the subquery (SELECT \* FROM TA WHERE TA.STUDENT\_ID = S.STUDENT\_ID) returns even a single row, the condition evaluates to TRUE.
* If TRUE, the corresponding student is selected.

### Why Use the EXISTS Clause?

* Efficiency:
  + Faster than the IN clause in certain scenarios.
  + Only checks if at least one row satisfies the condition instead of comparing against all values.
* Use Case:
  + Selects records when a condition matches at least one row.
* Syntax: WHERE EXISTS (SUBQUERY).
  + The EXISTS clause returns TRUE if the subquery returns at least one row; otherwise, it returns FALSE.



## Problem Statement: 7

* You are given a table STUDENTS with the following columns:
  + STUDENT\_ID: Unique identifier for each student.
  + NAME: Name of the student.
  + PSP: Performance Score Percentage of the student.
  + BATCH\_ID: ID of the batch the student belongs to.
* **Objective**: Return a table containing all columns (STUDENT\_ID, NAME, PSP, BATCH\_ID) and an additional column for the AVERAGE\_PSP **of the student's batch**.

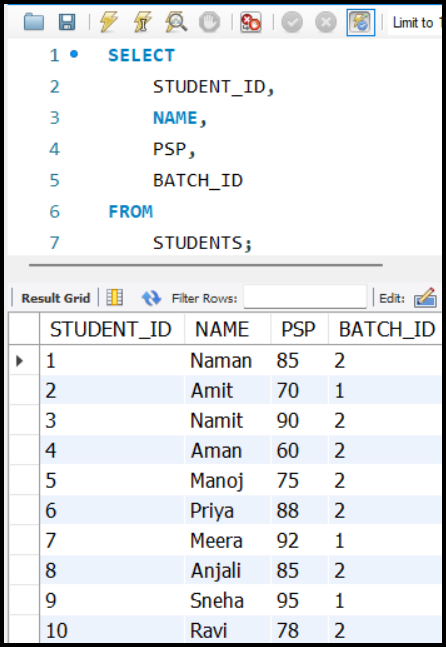
### **Challenges with** GROUPBY

1. **Why not use** GROUPBY**?**
   * In GROUP BY, only columns used in the aggregation or part of the GROUP BY clause can be included in the SELECT statement.
   * Aggregating AVERAGE PSP with non-aggregated columns like Name or ID isn’t directly possible.

### Subqueries in the SELECT Clause

Subqueries can exist in various clauses within a SQL query, including:

1. WHERE **Clause**
   * Seen in previous examples.
2. FROM **Clause**
   * Subqueries can be used to create temporary tables or views within a query.
3. SELECT **Clause**
   * Subqueries can compute values dynamically for each row being selected.



### Solution Using Subqueries in SELECT Clause

1. Basic Query Without AVERAGE PSP:

SELECT

    STUDENT\_ID,

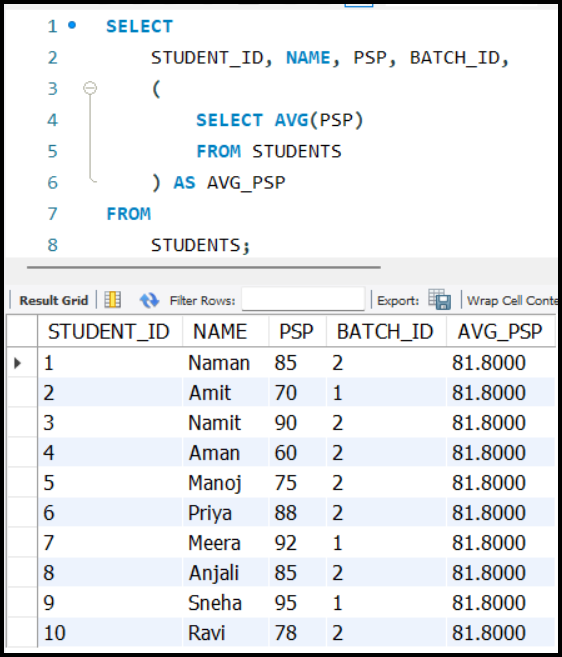
    NAME,

    PSP,

    BATCH\_ID

FROM

    STUDENTS;

1. Adding Average PSP of All Students Using a Subquery:

* Subquery computes the overall average PSP for all students:

SELECT

    STUDENT\_ID, NAME, PSP, BATCH\_ID,

    (

        SELECT AVG(PSP)

        FROM STUDENTS

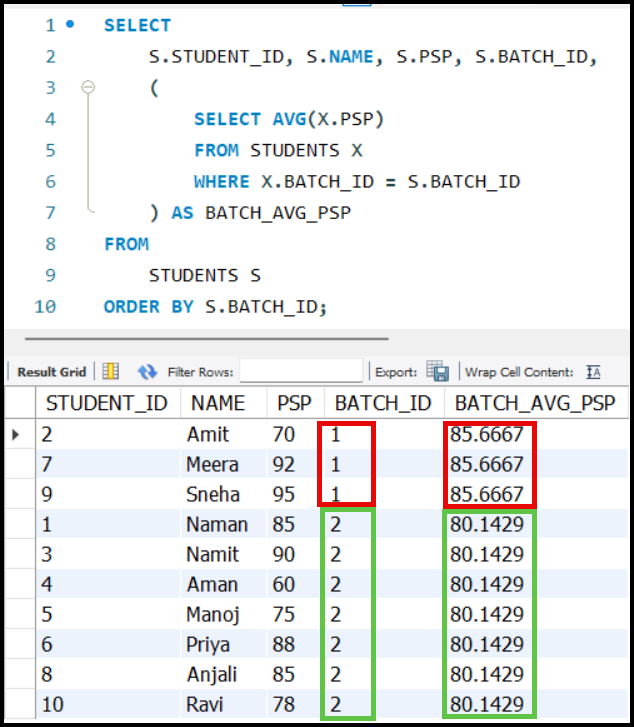
    ) AS AVG\_PSP

FROM

    STUDENTS;

* This adds a column showing the overall average PSP for every row.

1. Adding Average PSP of the Student’s Batch (Correlated Subquery):

* Why Correlation?
  + To compute the Average PSP specific to the batch of each student, the subquery must reference the current row of the outer query.
* Query with Correlation:

SELECT

    S.STUDENT\_ID, S.NAME, S.PSP,

    S.BATCH\_ID,

    (

        SELECT AVG(X.PSP)

        FROM STUDENTS X

        WHERE X.BATCH\_ID = S.BATCH\_ID

    ) AS BATCH\_AVG\_PSP

FROM

    STUDENTS S

ORDER BY S.BATCH\_ID;

* Explanation:
  + S: Alias for the outer query (STUDENTS table).
  + X: Alias for the inner subquery (STUDENTS table).
  + Correlation occurs through the condition X.BATCH\_ID = S.BATCH\_ID, ensuring that the subquery computes the average PSP only for STUDENTS in the same batch as the current row.

### **Types of Subqueries**

1. **Subqueries in** WHERE **Clause**:
   * These subqueries are used to filter rows based on conditions.
   * Example: SELECT \* FROM STUDENTS WHERE PSP > (SELECT AVG(PSP) FROM STUDENTS);
2. **Subqueries in** FROM **Clause**:
   * These subqueries can be used as temporary tables or derived tables.
   * Example: SELECT \* FROM (SELECT \* FROM STUDENTS) AS TEMP;
3. **Subqueries in** SELECT **Clause**:
   * These subqueries are used to calculate additional columns in the result set.
   * Example: SELECT ID, NAME, (SELECT AVG(PSP) FROM STUDENTS) AS AVG\_PSP FROM STUDENTS;

## Problem Statement: 8

* **Objective**: We have 3 tables STUDENTS, BATCHES and TEACHERS. Print the student name and the name of the teacher who is teaching that student.

### Step-by-Step Approach

1. **Join Query:**
   * Join STUDENTS, BATCHES, and TEACHERS tables.
   * Query:

SELECT

    S.STUDENT\_ID, S.STUDENT\_NAME,

    B.BATCH\_ID, B.BATCH\_NAME,

    T.TEACHER\_ID, T.TEACHER\_NAME

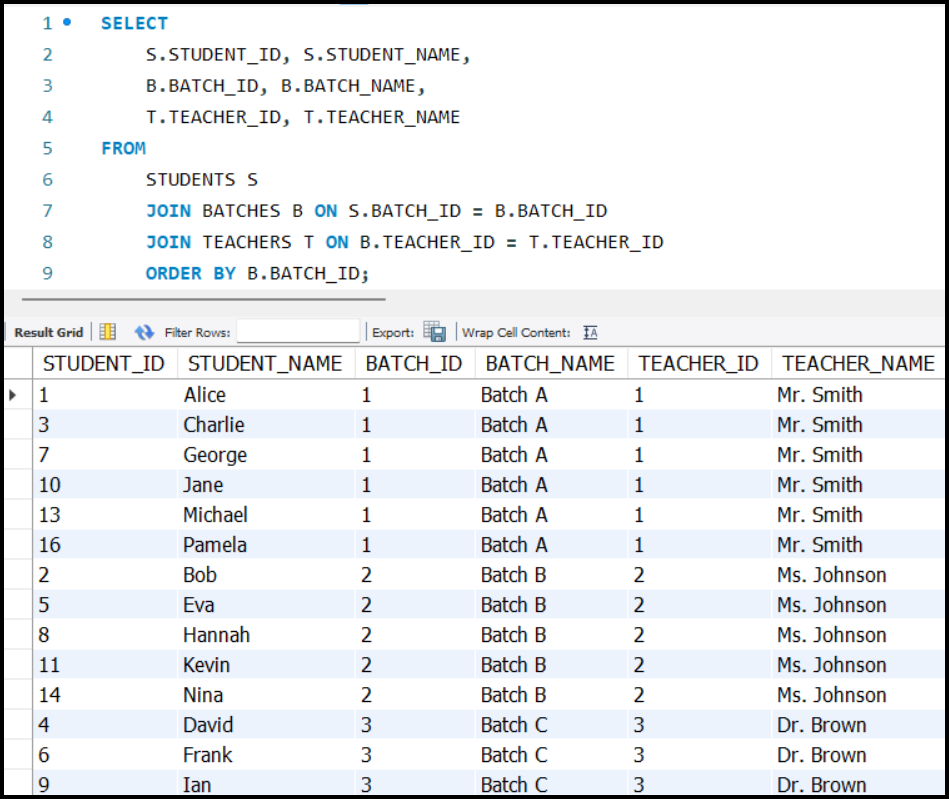
FROM

    STUDENTS S

    JOIN BATCHES B ON S.BATCH\_ID = B.BATCH\_ID

    JOIN TEACHERS T ON B.TEACHER\_ID = T.TEACHER\_ID

    ORDER BY B.BATCH\_ID;



1. **Create a Virtual Table:**

#### **Flattened Table Concept**

* A **flattened table** could simplify this process by including:
  + STUDENT\_ID, STUDENT\_NAME, BATCH\_ID, BATCH\_NAME, TEACHER\_ID, TEACHER\_NAME.
* Once this table is available, fetching the required data is simpler as shown below:

SELECT

    STUDENT\_NAME,

    TEACHER\_NAME

FROM

    FLATTENED\_TABLE;

* However, since this table doesn't exist initially, it can be created dynamically using a subquery.

#### **Creating a Flattened Table Using Subqueries**

* Use the 1st JOIN approach as subquery in the FROM clause:
* Query:

SELECT

    BIG\_JOIN\_TABLE.STUDENT\_NAME,

    BIG\_JOIN\_TABLE.TEACHER\_NAME

FROM

    (

        SELECT

            S.STUDENT\_ID, S.STUDENT\_NAME,

            B.BATCH\_ID, B.BATCH\_NAME,

            T.TEACHER\_ID, T.TEACHER\_NAME

        FROM

            STUDENTS S

            JOIN BATCHES B ON S.BATCH\_ID = B.BATCH\_ID

            JOIN TEACHERS T ON B.TEACHER\_ID = T.TEACHER\_ID

    ) AS BIG\_JOIN\_TABLE

    ORDER BY BATCH\_ID;

#### Explanation:

* The subquery creates a virtual table (BIG\_JOIN\_TABLE).
* The outer query selects specific columns (STUDENT\_NAME, TEACHER\_NAME).

#### Advantages of Subqueries in the FROM Clause

* **Modularity**:
  + Enables breaking down complex queries into smaller parts.
* **Reusability**:
  + The virtual table can be reused for different analyses.
    - E.g., Fetching student names with batch names or teacher names.
* **Ease of Maintenance**:
  + Easier to debug and modify complex SQL queries.
* **Scalability**:
  + If new requirements emerge, like fetching batch names, the existing subquery can be reused or extended.

#### Important Points

* **Naming Virtual Tables**:
  + Every subquery in the FROM clause must be assigned an alias.
    - Example: AS BIG\_JOIN\_TABLE.
* **Flattening Tables**:
  + Subqueries in the FROM clause are commonly used to flatten multiple joined tables into a single virtual table with multiple columns.
* **Use Cases**:
  + Often used for creating aggregated data or summarized views for analysis.
  + Simplifies queries when dealing with multi-layered joins or calculations.

# Views

## Problem with FORM Clause Subqueries

#### Subqueries in the FROM Clause

* **When are subqueries used in the** FROM **clause?**
  + Typically used when the same subquery needs to be reused multiple times in different queries for data analysis or execution purposes.
* **Example of usage:**
  + Analysts might copy a subquery and paste it into their queries repeatedly.

#### Problems with Copy-Pasting Subqueries

* **Repetition of Code:**
  + Subqueries are copy-pasted **across multiple queries, potentially hundreds of times, throughout the company.**
* **Maintenance Issues:**
  + **If the subquery needs to be updated (e.g., changing column names from** S.NAME **to** S.FIRST\_NAME **or** S.LAST\_NAME**), updates must be made in all occurrences.**
  + **This creates a massive effort and increases the risk of:**
    - **Human errors: Missing updates in some queries.**
    - **Broken queries: Resulting from outdated subquery references.**

#### **Key** Challenge**:**

* **Lack of centralized management for shared query logic.**

## Problems with Creating Actual Big Fat Tables

* Instead of having a subquery, we can create a BIG\_FAT\_TABLE.

#### Option 1: Replace Original Tables with a BIG\_FAT\_TABLE

* Disadvantages:
  + Denormalization introduces redundancy and anomalies.
  + Violates principles of database normalization.
  + Issues with:
    - Updates
    - Deletes
    - Data consistency

#### Option 2: Maintain Both Original Tables and BIG\_FAT\_TABLE

* Disadvantages:
  + Storage Overhead: Duplication of data increases storage requirements.
  + Consistency Issues: Changes in original tables may not be reflected in the big fat table unless properly synchronized.

## Introduction to Views

* A view is a virtual table created based on the result-set of an SQL query.
* It does not physically store the data but provides a way to simplify complex queries and enhance code reusability, centralized query and avoids redundancy.

### **Advantages of Views**

* **Simplification**: Views simplify complex queries by encapsulating them into a single virtual table.
* **Reusability**: Once created, views can be used in other queries without the need to rewrite the complex logic.
* **Maintenance**: Any change in the view definition is reflected in all queries using the view, reducing maintenance overhead.
* **Consistency**: Ensures consistent results by providing a single source of truth.

## Problem Statement

* **Objective**: Create a view that provides combined data from multiple tables (ORDERS, ORDER\_ITEMS, PRODUCTS, CUSTOMERS).
* **Required Columns**:

ORDER\_ID

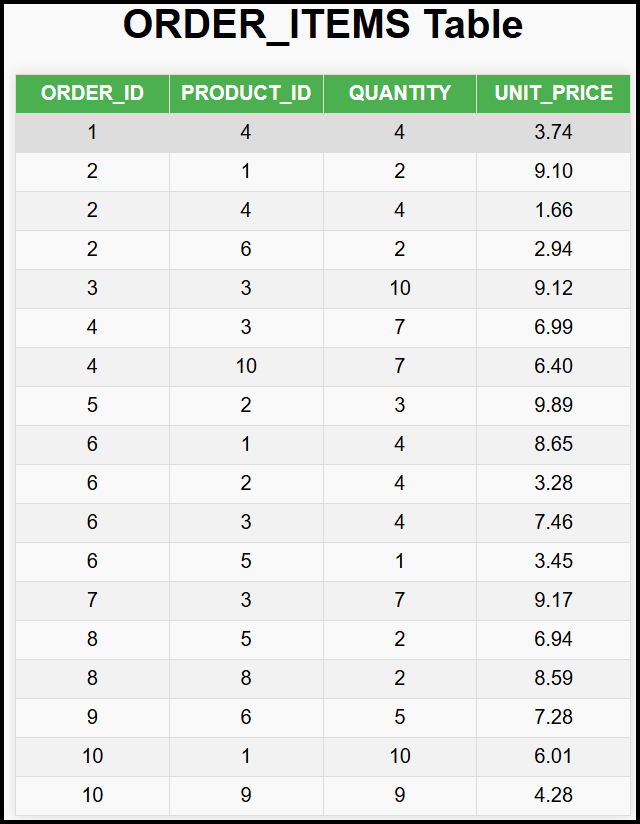
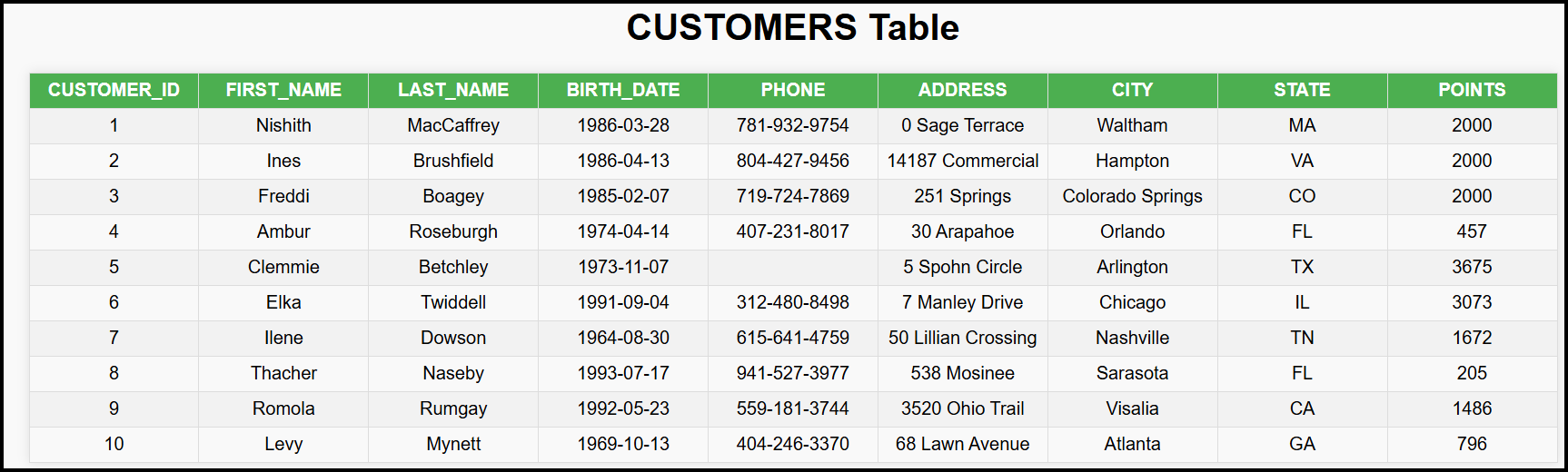
CUSTOMER\_ID

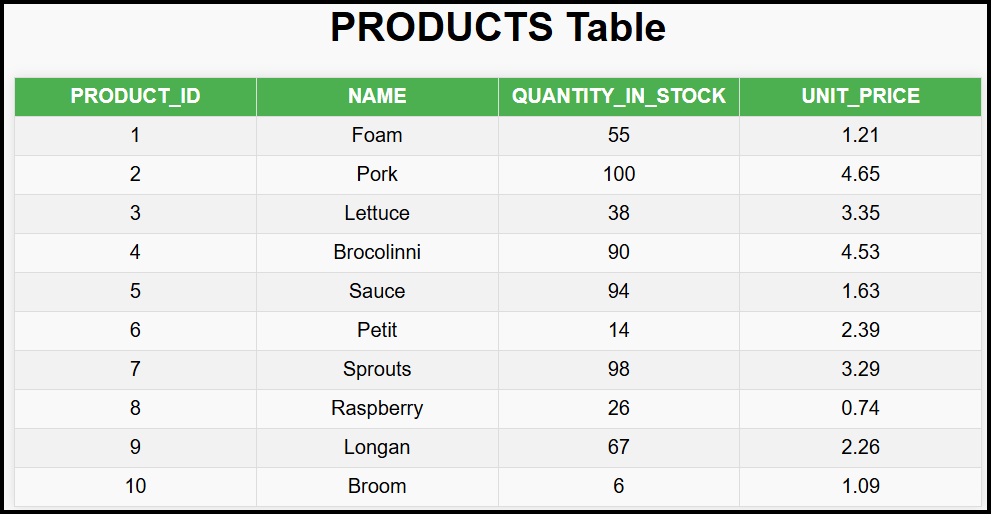
CUSTOMER\_NAME

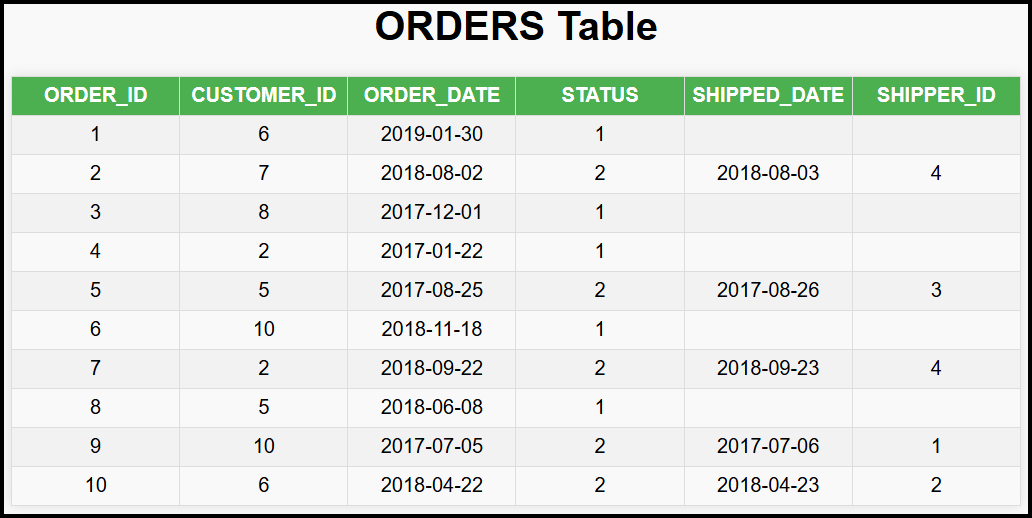
PRODUCT\_ID

PRODUCT\_NAME

QUANTITY





#### SQL Join Logic:

* To combine data from the above tables, JOINs are performed:
  + ORDERS table joined with ORDER\_ITEMS on ORDER\_ID.
  + ORDER\_ITEMS table joined with PRODUCTS on PRODUCT\_ID.
  + ORDERS table joined with CUSTOMERS on CUSTOMER\_ID.
* SQL Query:

SELECT

    O.ORDER\_ID,

    C.CUSTOMER\_ID,

    C.FIRST\_NAME AS CUSTOMER\_NAME,

    OI.PRODUCT\_ID,

    P.NAME AS PRODUCT\_NAME,

    OI.QUANTITY

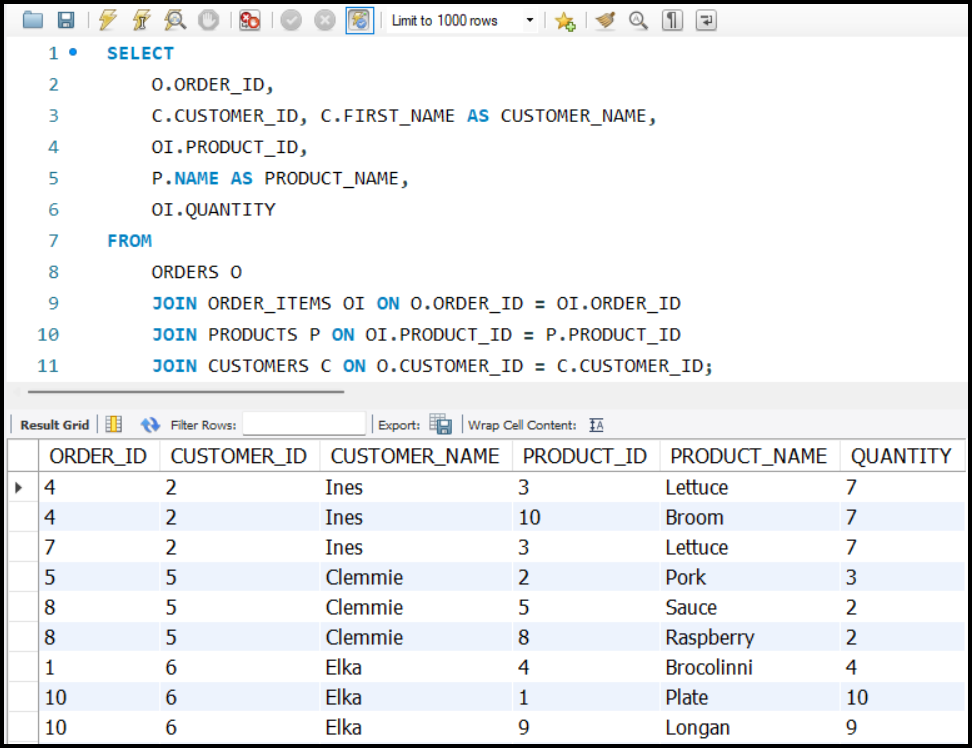
FROM

    ORDERS O

    JOIN ORDER\_ITEMS OI ON O.ORDER\_ID = OI.ORDER\_ID

    JOIN PRODUCTS P ON OI.PRODUCT\_ID = P.PRODUCT\_ID

    JOIN CUSTOMERS C ON O.CUSTOMER\_ID = C.CUSTOMER\_ID;



### Creating a View

* A view is created using the CREATE VIEW statement:

CREATE VIEW BIG\_FAT\_VIEW AS

SELECT

    O.ORDER\_ID,

    C.CUSTOMER\_ID,

    C.FIRST\_NAME AS CUSTOMER\_NAME,

    OI.PRODUCT\_ID,

    P.NAME AS PRODUCT\_NAME,

    OI.QUANTITY

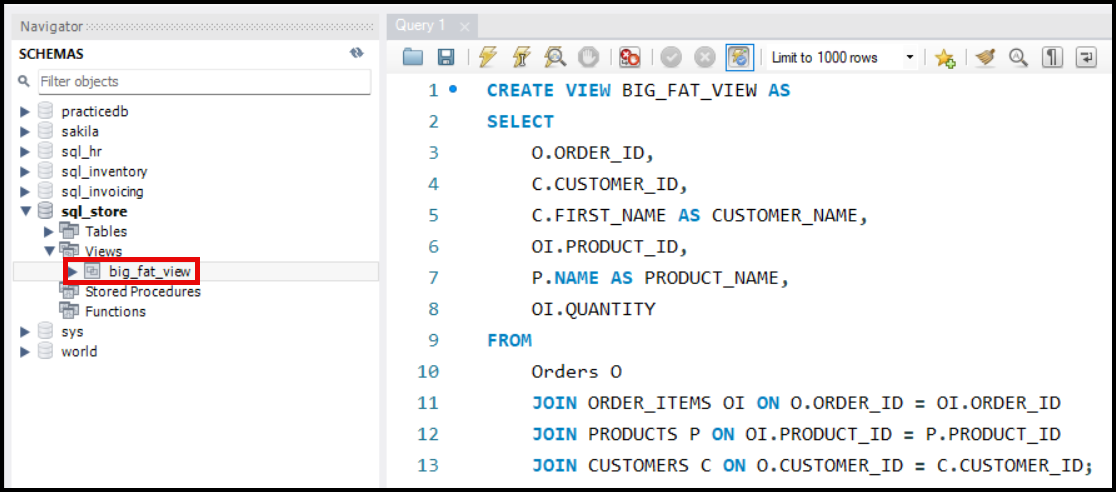
FROM

    Orders O

    JOIN ORDER\_ITEMS OI ON O.ORDER\_ID = OI.ORDER\_ID

    JOIN PRODUCTS P ON OI.PRODUCT\_ID = P.PRODUCT\_ID

    JOIN CUSTOMERS C ON O.CUSTOMER\_ID = C.CUSTOMER\_ID;



### Accessing the View:

* The view acts like a table and can be queried directly:

SELECT

    BF.ORDER\_ID,

    BF.CUSTOMER\_NAME

FROM

    BIG\_FAT\_VIEW BF;

#### View Behaviour:

* **No Data Storage**: A view does not store data; it dynamically runs the underlying query when accessed.
* **Dependency**: If a table referenced in the view is deleted or altered, the view will start erroring out.
* **Automatic Updates**: Any update in the original tables reflects automatically in the view.

#### Advantages of Views:

* Avoids duplication of queries.
* Centralizes logic for easier updates.
* Enhances readability and usability for analysts and clients.

#### Limitations of Views:

* Views are dependent on the underlying tables.
* Complex views can sometimes degrade performance.

### One more example:

* Create view for the STUDENTS, BATCHES and TEACHERS table for the above discussed scenario.

CREATE VIEW STB\_VIEW AS

SELECT

    S.STUDENT\_ID, S.STUDENT\_NAME,

    B.BATCH\_ID, B.BATCH\_NAME,

    T.TEACHER\_ID, T.TEACHER\_NAME

FROM

    STUDENTS S

    JOIN BATCHES B ON S.BATCH\_ID = B.BATCH\_ID

    JOIN TEACHERS T ON B.TEACHER\_ID = T.TEACHER\_ID;

