Aggregate Queries

Contents

[Introduction 2](#_Toc186492732)

[Group By and Aggregate Functions 6](#_Toc186492733)

[Understanding Aggregate Functions and Queries 17](#_Toc186492734)

[Having Clause 21](#_Toc186492735)

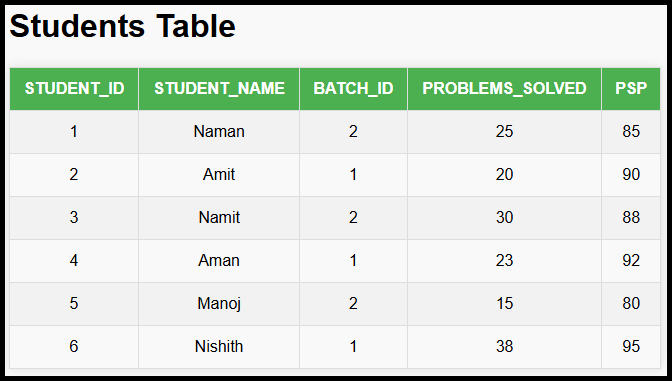
[Pseudocode to simulate SQL query processing 25](#_Toc186492736)

## **Introduction**

* Traditional SQL queries work row by row, selecting rows based on conditions.
* Aggregate queries differ as they produce a result by combining multiple rows into a **single output**.

#### **Use Case of Aggregate Queries**

* Example:
  + **Students Table**  
    Columns: STUDENT\_ID, STUDENT\_NAME, PROBLEMS\_SOLVED (number of problems solved by each student).



* + Task: Calculate the **average number of problems solved**.
* Row-by-row processing cannot handle this because the result is a computation over multiple rows.

#### **Aggregate Functions in SQL**

SQL provides several aggregate functions:

* MAX(): Maximum value of a column.
* MIN(): Minimum value of a column.
* AVG(): Average value of a column.
* SUM(): Sum of values in a column.
* COUNT(): Count of rows or non-null values.

#### **Key Concepts**

1. **Processing Aggregate Functions**:
   * Aggregate functions are applied after the virtual table is created by the FROM clause.
   * Example:

SELECT MAX(PROBLEMS\_SOLVED) FROM STUDENTS;

* + - Combines all rows to find the maximum value in PROBLEMS\_SOLVED.

1. **Order of SQL Execution**:
   * Traditional Query:
     + SELECT
     + FROM
     + WHERE
     + ORDER BY
   * Aggregate queries introduce new clauses such as GROUP BY (discussed later).
2. **Rules for Using Aggregate Functions**:
   * Queries cannot mix aggregate and non-aggregate columns in SELECT without grouping.
     + Example (invalid):

SELECT

    STUDENT\_NAME,

    MAX(PROBLEMS\_SOLVED)

FROM

    STUDENTS;

* + - * Ambiguous because STUDENT\_NAME doesn’t have a meaningful relationship with the aggregated value.

1. **Combining Multiple Aggregates**:
   * Example Query:

SELECT

    COUNT(STUDENT\_NAME),

    MAX(PROBLEMS\_SOLVED),

    MIN(PROBLEMS\_SOLVED)

FROM

    STUDENTS;

* + - Outputs:
      * Total number of students (COUNT).
      * Maximum problems solved (MAX).
      * Minimum problems solved (MIN).

1. **Handling** NULL **Values**:
   * Aggregate functions ignore NULL values.
   * Example:
     + If a row has a NULL in the Name column:
       - COUNT(Name) excludes that row.
       - MAX and MIN functions still compute based on non-NULL values.
   * Best Practices:
     + Use COUNT(\*) instead of COUNT(column) to include all rows.
     + Use non-nullable columns like ID for accurate results:

SELECT COUNT(ID) FROM STUDENTS;

#### **Best Practices**

1. **Using** COUNT(\*):
   * Ensures all rows are considered as rows themselves cannot be NULL.
2. **Non-Nullable Columns**:
   * Prefer non-nullable columns (e.g., ID) to avoid missing rows in aggregation.

#### **Summary of Aggregate Functions**

* Aggregate functions process multiple rows to produce a single output value.
* Common aggregate functions include:
  + COUNT: Number of rows or non-null values.
  + MAX: Maximum value.
  + MIN: Minimum value.
  + SUM: Total sum.
  + AVG: Average of values.

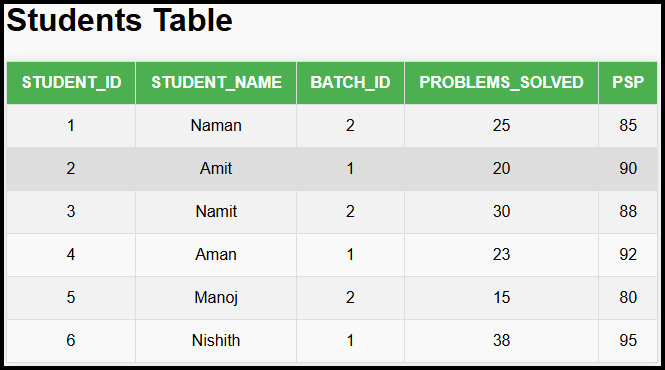
## **Group By and Aggregate Functions**

#### **Objective**

* Calculate specific values (e.g., maximum or average PSP) grouped by a column in a table.
* Example scenario: Find the **maximum PSP** or **average PSP** for each batch in a STUDENTS table.

#### **Key Concepts**

1. **Table Structure**
   * Columns: STUDENT\_ID, STUDENT\_NAME, BATCH\_ID, PROBLEMS\_SOLVED, PSP
   * Example data:



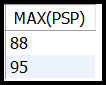
1. **When to Use Aggregate Functions**
   * Aggregate functions (e.g., MAX, AVG) are used to **combine rows** to compute a single value or a set of values.
   * Example:
     + To find the **maximum PSP** in the entire table:

SELECT MAX(PSP) FROM STUDENTS;

1. **When Aggregate Functions Alone Are Not Enough**
   * Aggregate functions compute a **single value** for all rows.
   * When we need values grouped by a specific column (e.g., BATCH\_ID), **grouping** is required.
2. **Group By Clause**
   * The GROUPBY clause allows aggregation to be performed on subsets of rows.
   * Steps:
     1. **Group Rows**: Rows with the same value in the specified column(s) are grouped together.
     2. **Apply Aggregate Function**: Compute the aggregate function (e.g., MAX, AVG) on each group.

#### **Example Problem**

* Find the **maximum PSP** for each batch:
  + Expected Output:



#### **Query Explanation**

1. **Query**:

SELECT

    MAX(PSP)

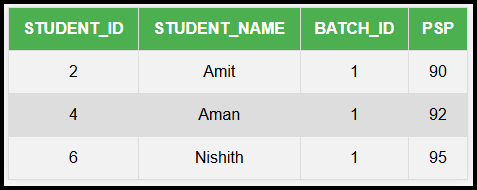
FROM

    STUDENTS

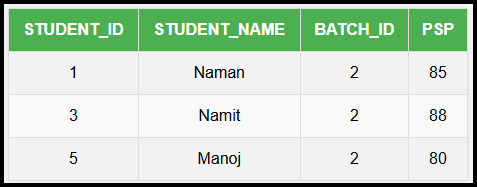
GROUP BY

    BATCH\_ID;

1. **Steps of Execution**:
   * **Step 1**: Group rows by BATCH\_ID:
     + Group 1 (Batch ID = 1):

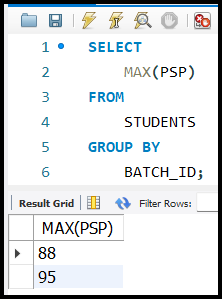


* + - Group 2 (Batch ID = 2):



* + **Step 2**: Apply MAX(PSP) to each group:
    - Group 1: MAX(90, 92, 95) = 95
    - Group 2: MAX(85, 88, 80) = 88

1. **Result**:



#### **Group By Clause Behavior**

1. **Syntax**:

SELECT

    COLUMN1,

    AGGREGATE\_FUNCTION(COLUMN2)

FROM

    TABLE\_NAME

GROUP BY

    COLUMN1;

* + COLUMN1: Column used for grouping rows.
  + AGGREGATE\_FUNCTION: Function applied to each group (e.g., MAX, AVG).

1. **Execution Flow**:
   * Step 1: Group rows based on GROUP BY column.
   * Step 2: Apply aggregate function to each group.
2. **Result**: Outputs one row per group.

#### **Notes on Aggregate Function Usage with Group By**

* **Multiple Groups**:
  + Aggregate functions are computed **independently for each group**.
  + Example:

SELECT

    BATCHID,

    AVG(PSP)

FROM

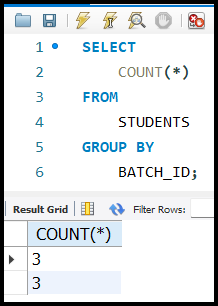
    STUDENTS

GROUP BY

    BATCHID;

* **Virtual Tables**:
  + Grouping creates intermediate "virtual tables" for each group before aggregation.

#### Aggregation in SQL

1. **Counting Students by Batch**
   * Query:

SELECT

    COUNT(\*)

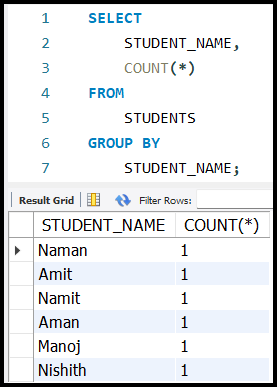
FROM

    STUDENTS

GROUP BY

    BATCH\_ID;

* + Explanation:  
    Groups students by their BATCH\_ID and counts the number of students in each batch.

1. **Counting Students by Name**
   * Query:

SELECT

    STUDENT\_NAME,

    COUNT(\*)

FROM

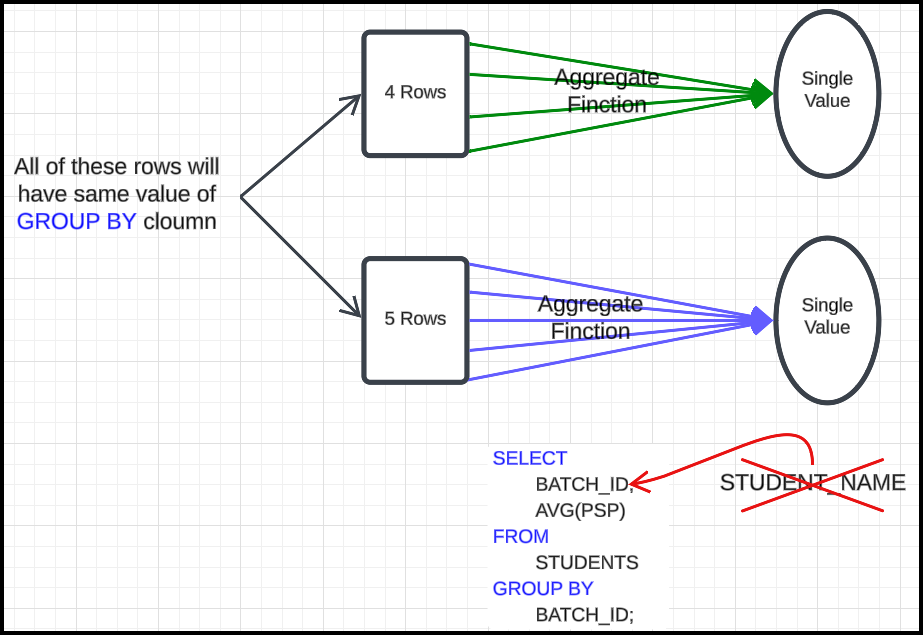
    STUDENTS

GROUP BY

    STUDENT\_NAME;

* + Explanation:  
    Groups students by their STUDENT\_NAME and counts how many students share the same name.

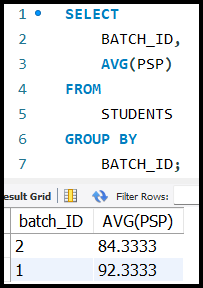
1. **Key Concepts for SELECT with GROUP BY**:



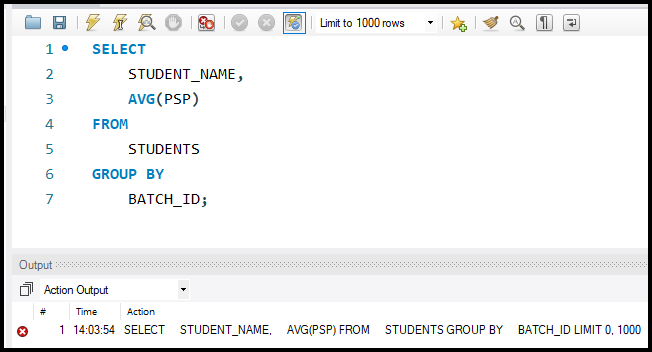
* + SELECT statement can have:
    - All aggregate functions (e.g., COUNT, AVG, etc.).
    - All non-aggregate columns.
    - Aggregate functions + columns included in the GROUP BY clause.
  + Columns in the GROUP BY clause will have the same values for rows grouped together.

#### Aggregation with Additional Columns

* Example:  
  To get BATCH\_ID along with the average PSP (some column):



* Explanation:  
  All rows grouped under the same BATCH\_ID contribute to the calculated average.
* **Invalid Query**:



* + This is invalid as STUDENT\_NAME is neither an aggregate function nor included in the GROUP BY clause.

#### Joining Tables for Aggregation

1. **Use Case**:

Combine data from multiple tables, such as STUDENTS and BATCHES, to calculate average PSP for each batch name.

1. **Steps**:
   * Create a virtual table using a JOIN.
   * Use the virtual table to perform the required aggregation.
2. **Query Example**:

SELECT

    B.BATCH\_NAME AS BATCH\_NAME,

    AVG(S.PSP) AS AVERAGE\_PSP

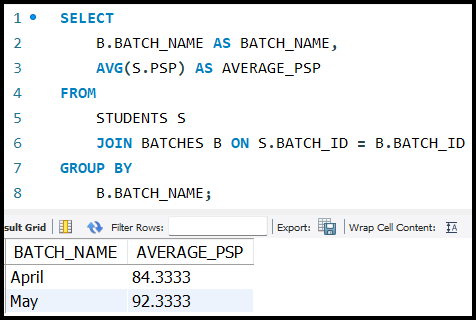
FROM

    STUDENTS S

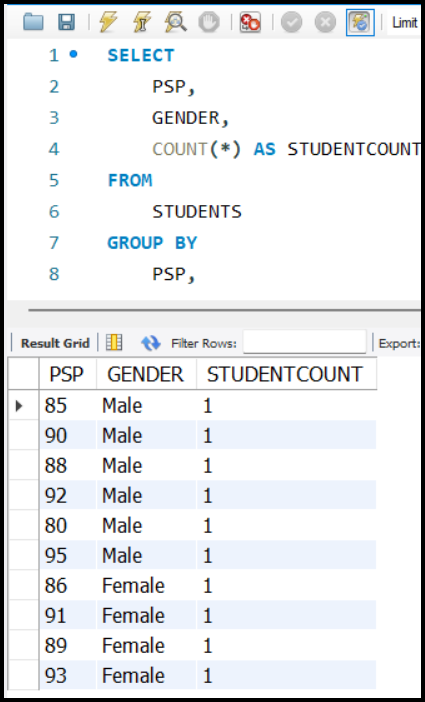
    JOIN BATCHES B ON S.BATCH\_ID = B.BATCH\_ID

GROUP BY

    B.BATCH\_NAME;

1. **Explanation**:
   * The JOIN combines the STUDENTS and BATCHES tables based on the condition S.BATCH\_ID = B.BATCH\_ID.
   * GROUP BY B.BATCH\_NAME to calculate the average PSP for each batch.

#### Best Practices

* Write the SELECT clause last to simplify query construction.
* Think of a JOIN as creating a virtual table.
* Approach SQL problems step by step:
  1. Create a virtual table (JOIN).
  2. Apply aggregation (GROUP BY).
  3. Select the desired output.

**Using Multiple Columns in GROUP BY**:

* Yes, you can have multiple columns in the GROUP BY clause. It essentially groups rows based on the unique combinations of values across those columns. For example…
* This groups the data first by PSP and then further by GENDER, producing counts for each combination.

**Non-Aggregate Columns in SELECT with GROUP BY**:

* SQL requires that any non-aggregated column in the SELECT clause must either be part of the GROUP BY clause or be used in an **aggregate function**. This ensures a logical association between grouped rows and the output.

**Behaviour Without GROUP BY**:

* Without a GROUP BY, aggregate functions like SUM or COUNT operate on the entire result set, returning a single aggregated value. Example:

SELECT

    COUNT(\*)

FROM

    ORDERS;

**Joining Tables and Aggregating**:

* When aggregating data from joined tables, ensure the grouping makes sense within the joined context. For example:

SELECT

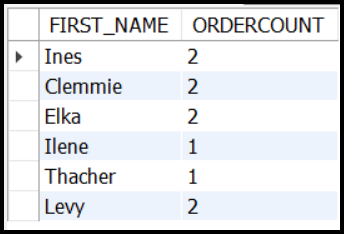
    C.FIRST\_NAME,

    COUNT(O.ORDER\_ID) AS ORDERCOUNT

FROM

    CUSTOMERS C

    JOIN ORDERS O ON C.CUSTOMER\_ID = O.CUSTOMER\_ID

GROUP BY

    C.CUSTOMER\_ID,

    C.FIRST\_NAME;

**Ambiguity with Common Names**:

* If multiple rows share a common name but belong to different entities (e.g., customers with the same name), grouping only by FirstName may combine unrelated data. Including a unique identifier like CUSTOMER\_ID in GROUP BY ensures accurate results.

**Error with Non-Aggregate Columns Not in GROUP BY**:

* SQL enforces that all selected columns, other than those in aggregate functions, must be listed in the GROUP BY clause. Failing to do so results in errors like:

*Column 'X' is invalid in the select list because it is not contained in either an aggregate function or the GROUP BY clause.*

**Output Presentation**:

* To avoid ambiguity or incorrect results, include all relevant columns in the GROUP BY. If extra details are needed (e.g., combining names and IDs), the grouping should align accordingly:

SELECT

    C.CUSTOMER\_ID,

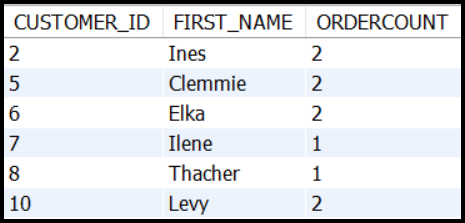
    C.FIRST\_NAME,

    COUNT(O.ORDER\_ID) AS ORDERCOUNT

FROM

    CUSTOMERS C

    JOIN ORDERS O ON C.CUSTOMER\_ID = O.CUSTOMER\_ID

GROUP BY

    C.CUSTOMER\_ID,

    C.FIRST\_NAME;

## Understanding Aggregate Functions and Queries

1. **Aggregate Functions**: Functions that perform calculations on a set of rows and return a single value. Common examples:
   * SUM: Calculates the total.
   * AVG: Computes the average.
   * COUNT: Counts the number of rows.
2. **Aggregate Query**: A query that uses aggregate functions.

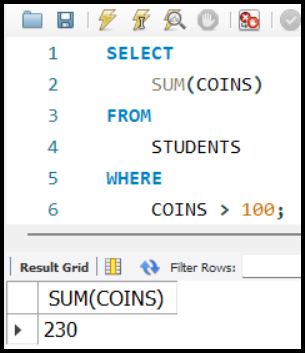
#### Process Flow in SQL

1. **Basic Aggregation**:
   * Example:

SELECT SUM(COINS) FROM STUDENTS;

* + - Aggregates all rows in the table and returns the total sum of the COINS column.



1. **Filtered Aggregation**:
   * Filtering rows using WHERE before applying an aggregate function.
   * Example:

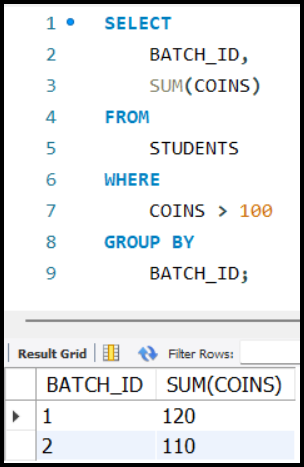
SELECT SUM(COINS) FROM STUDENTS WHERE COINS > 100;

* + - Filters rows where COINS > 100 and sums those values.
    - Output: 230 (sum of 120 and 110 from rows where COINS > 100).

#### Aggregation with Groups

1. **Grouping Rows**:
   * Use GROUP BY to divide rows into subsets based on a column value.
   * Example Table:



* + Query:

SELECT

    BATCH\_ID,

    SUM(COINS)

FROM

    STUDENTS

WHERE

    COINS > 100

GROUP BY

    BATCH\_ID;

* + - Steps:
      1. **Filter Rows**: Include only rows WHERE COINS > 100.
      2. **Group Rows by** BATCH\_ID: Create subsets based on BATCH\_ID.
      3. **Apply Aggregation**: Calculate the sum for each group.

#### Execution Order in SQL

1. WHERE **Clause**: Filters rows before aggregation.
2. GROUPBY **Clause**: Groups filtered rows into subsets.
3. **Aggregate Function**: Applied to groups or entire result set.

#### Programming Analogy

* **Steps for SQL Execution**:
  1. **Filter Rows**:
     + Check each row in the table; if it matches the WHERE condition, add it to the answer.
  2. **Group Rows (if** GROUPBY**)**:
     + Divide rows into groups based on the specified column(s).
  3. **Apply Aggregate Function**:
     + Perform aggregation on all rows (if no GROUP BY) or on each group (if GROUP BY exists).
* **Pseudocode**:

|  |
| --- |
| answer []  for row in table:      if row matches condition of WHERE:          add row to answer  if aggregate query:      if GROUP BY:          group rows in answer into sub\_lists          for each sub\_list:              compute aggregate and output      else:          compute aggregate on all rows in answer  else:      print columns in SELECT |

#### Key Differences

1. **Aggregated Query vs Aggregate Function**:
   * Aggregate queries: Queries that use aggregate functions.
   * Aggregate functions: The functions themselves (e.g., SUM, AVG).

## Having Clause

#### Problem Statement

* We have a **STUDENTs table** with the following columns:



**Objective**: For every batch:

1. Select students with **PSP > 70**.
2. Calculate the average PSP for these students in the batch.
3. Only output batches where the **average PSP > 85**.

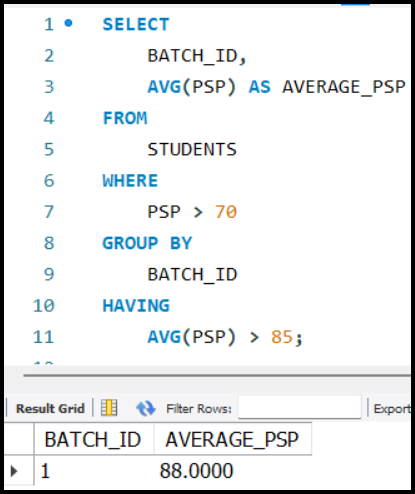
#### Calculations with Example Data

1. **Batch 1**:
   * Students with PSP > 70: Aman (92), Nishith (95), Sneha (77)
   * Average PSP:
   * **Output Batch 1** (Average PSP > 85).
2. **Batch 2**:
   * Students with PSP > 70: Naman (85), Manoj (72), Priya (86), Anjali (81)
   * Average PSP:
   * **Exclude Batch 2** (Average PSP <= 85).

### **Steps to Solve the Problem**

1. **Filter Rows**: Use WHERE clause to filter rows where PSP > 70.
2. **Group Data**: Use GROUP BY clause to group students by their batch (BATCH\_ID).
3. **Aggregate Data**: Compute the average PSP for each batch using the AVG() function.
4. **Filter Aggregated Results**: Use HAVING clause to filter batches with AVG(PSP) > 85.

### **SQL Query**

The SQL query to achieve this is:

SELECT

    BATCH\_ID,

    AVG(PSP) AS AVERAGE\_PSP

FROM

    STUDENTS

WHERE

    PSP > 70

GROUP BY

    BATCH\_ID

HAVING

    AVG(PSP) > 85;

### **Explanation of the Query**

1. **WHERE PSP > 70**:
   * Filters rows to include only students with PSP > 70.
2. **GROUP BY BATCH\_ID**:
   * Groups these filtered rows by BATCH\_ID.
3. **AVG(PSP)**:
   * Calculates the average PSP for each group (batch).
4. **HAVING AVG(PSP) > 85**:
   * Filters the aggregated results to include only batches with average PSP > 85.

### **WHERE vs HAVING**

1. **Purpose of** WHERE **Clause:**
   * The WHERE clause is used to filter rows based on specific conditions.
   * It operates on individual rows **before aggregation** occurs.
2. **Limitation of** WHERE**:**
   * WHERE cannot be used to filter aggregated results (e.g., averages, sums, or counts).
   * Aggregation involves combining rows to produce a summary (e.g., AVG(column\_name)), which WHERE cannot handle since it filters only row-level conditions.
3. **Why** WHERE **Fails for Aggregations:**
   * Example: Filtering AVG(PSP) > 85 cannot be done using WHERE because AVG(PSP) is an **aggregated value**, not a property of an individual row.
4. **Solution: Use the** HAVING **Clause:**
   * The HAVING clause is specifically designed for filtering aggregated data.
   * It operates **after aggregation**, allowing conditions to be applied to aggregated results.
5. **Syntax for** HAVING**:**

SELECT

    COLUMN\_NAME,

    AGGREGATE\_FUNCTION(COLUMN\_NAME)

FROM

    TABLE\_NAME

WHERE

    ROW\_CONDITION

GROUP BY

    COLUMN\_NAME

HAVING

    AGGREGATE\_FUNCTION(COLUMN\_NAME) CONDITION;

1. **Key Difference Between** WHERE **and** HAVING**:**
   * WHERE: Filters rows **before grouping/aggregation**.
   * HAVING: Filters aggregated data **after grouping/aggregation**.

**Example Query Explanation:** To find batches with an average PSP greater than 85:

1. Use WHERE PSP > 70 to filter rows based on individual PSP values.
2. Use GROUP BY BATCH\_ID to group students by their batch.
3. Use HAVING AVG(PSP) > 85 to filter groups based on the aggregated PSP.

SELECT BATCH\_ID, AVG(PSP) AS AVG\_PSP

FROM STUDENTS

WHERE PSP > 70

GROUP BY BATCH\_ID

HAVING AVG(PSP) > 85;

## Pseudocode to simulate SQL query processing

|  |
| --- |
| # Input: table (list of rows), where\_condition, select\_columns,  # aggregate\_function, group\_by\_column, having\_condition  # Step 1: Filter rows based on WHERE condition  answer = []  for row in table:      if row\_matches\_condition(row, where\_condition):          answer.append(row)  # Step 2: Check if it is an aggregated query  if not is\_aggregated\_query(select\_columns):      # For non-aggregated query: Print the selected columns      for row in answer:          print(select\_columns\_from\_row(row, select\_columns))  else:      # Step 3: Process aggregated queries      if group\_by\_column is None:          # No GROUP BY: Apply aggregate function over all rows in 'answer'          result = apply\_aggregate\_function(answer, aggregate\_function)          print(result)      else:          # With GROUP BY: Create subgroups and process each group          subgroups = group\_rows\_by\_column(answer, group\_by\_column)          final\_output = []          for group in subgroups:              # Compute aggregate function for the group              agg\_result = compute\_aggregate\_function(group, aggregate\_function)              # Check if the result matches the HAVING condition              if matches\_condition(agg\_result, having\_condition):                  final\_output.append(agg\_result)            # Print the final output for matching subgroups          for result in final\_output:              print(result) |

### Key Functions Explained:

1. **row\_matches\_condition(row, condition):**
   * Checks if a single row satisfies the WHERE condition.
2. **is\_aggregated\_query(select\_columns):**
   * Determines if the query involves an aggregate function.
3. **select\_columns\_from\_row(row, select\_columns):**
   * Extracts specified columns from a single row for SELECT in non-aggregated queries.
4. **apply\_aggregate\_function(rows, aggregate\_function):**
   * Applies an aggregate function (e.g., SUM, AVG, etc.) to all rows when there's no GROUP BY.
5. **group\_rows\_by\_column(rows, group\_by\_column):**
   * Divides rows into groups based on the GROUP BY column.
6. **compute\_aggregate\_function(group, aggregate\_function):**
   * Computes the aggregate function for a specific group of rows.
7. **matches\_condition(agg\_result, having\_condition):**
   * Checks if the aggregated result satisfies the HAVING condition.

### Example Walkthrough:

SELECT

    DEPARTMENT,

    AVG(SALARY)

FROM

    EMPLOYEES

WHERE

    AGE > 30

GROUP BY

    DEPARTMENT

HAVING

    AVG(SALARY) > 50000;

1. **Filter rows with** WHERE**: Keep employees** WHEREAGE > 30**.**
2. **Group rows by** DEPARTMENT**.**
3. **Compute** AVG(SALARY) **for each department.**
4. **Filter groups with** HAVINGAVG(SALARY) > 50000**.**
5. **Print the department and the average salary for matching groups.**

### Joins and Aggregation

#### Problem Statement:

Print the names of INSTRUCTORS whose **toppers** (students with PSP > 60) have an average **PSP > 80**.

* **Toppers**: Students with PSP > 60.
* Final Output: Names of INSTRUCTORS.

#### Step-by-Step Breakdown:

1. **Understanding the Tables**:

**Students Table**:

Contains columns: ID, NAME, INSTRUCTOR\_ID, PSP.

**Instructors Table**:

Contains columns: ID, NAME.

1. **Desired Table for Easier Query**:
   * A combined table of students and their instructors would simplify the process.
   * This can be achieved using a **JOIN** between the two tables.

#### Steps to Solve the Problem:

* **Join the Tables**:
  + Combine STUDENTS and INSTRUCTORS tables based on INSTRUCTOR\_ID:

SELECT

    \*

FROM

    STUDENTS AS S

    JOIN INSTRUCTORS AS I ON S.INSTRUCTORID = I.ID;

* **Filter Rows**:
  + Only select rows where PSP > 60 (to get toppers):

WHERE S.PSP > 60;

* **Group the Data**:
  + Group the filtered rows by INSTRUCTOR\_ID and INSTRUCTOR\_NAME.
  + Grouping helps calculate the average **PSP** for each instructor’s toppers.
* **Apply Aggregation**:
  + Compute the average **PSP** for each group:

HAVING AVG(S.PSP) > 80;

* + Only keep groups where the average **PSP** of toppers exceeds 80.
* **Select the Desired Columns**:
  + Output only the INSTRUCTOR\_NAME for matching groups:

SELECT I.NAME;

* **Final** **Query**:

SELECT I.NAME

FROM

    STUDENTS AS S

    JOIN INSTRUCTORS AS I ON S.INSTRUCTORID = I.ID

WHERE

    S.PSP > 60

GROUP BY

    I.ID, I.NAME

HAVING

    AVG(S.PSP) > 80;

1. **Join Tables**: Combine STUDENTS and INSTRUCTORS using INSTRUCTORID.
2. **Filter Toppers**: Use WHERE S.PSP > 60 to filter toppers.
3. **Group Toppers by Instructor**: Use GROUP BY I.ID, I.NAME to group toppers by their instructors.
4. **Filter Groups with Average PSP > 80**: Use HAVING AVG(S.PSP) > 80 to keep only the relevant groups.
5. **Output Instructor Names**: Use SELECT I.NAME to display only the instructor names.