**Madiha Aimon Tappal**

[**madihaaimon@gmail.com**](mailto:madihaaimon@gmail.com)

**Day - 6**

**Data Engineering Batch – 1**

**What Is the PARTITION BY Clause in SQL?**

The SQL PARTITION BY expression is a subclause of the OVER clause, which is used in almost all invocations of window functions like AVG (), MAX (), and RANK ().

Window functions operate on window frames which are sets of rows that can be different for each record in the query result. This is where the SQL PARTITION BY subclause comes in: it is used to define which records to make part of the window frame associated with each record of the result. The **OVER** and **PARTITION BY** clauses are commonly used with window functions in SQL. These clauses help in defining the window of rows for each calculation performed by the window function.

The first thing to focus on is the syntax. Here’s how to use the SQL PARTITION BY clause:

SELECT

    <column>,

    <window function=""> OVER (PARTITION BY <column> [ORDER BY <column>])

FROM table;

</column></column></window></column>

#### **Example 1: Using** OVER **and** PARTITION BY **for Running Total:**

-- Calculate the running total of age for each grade partition

SELECT

student\_id,

first\_name,

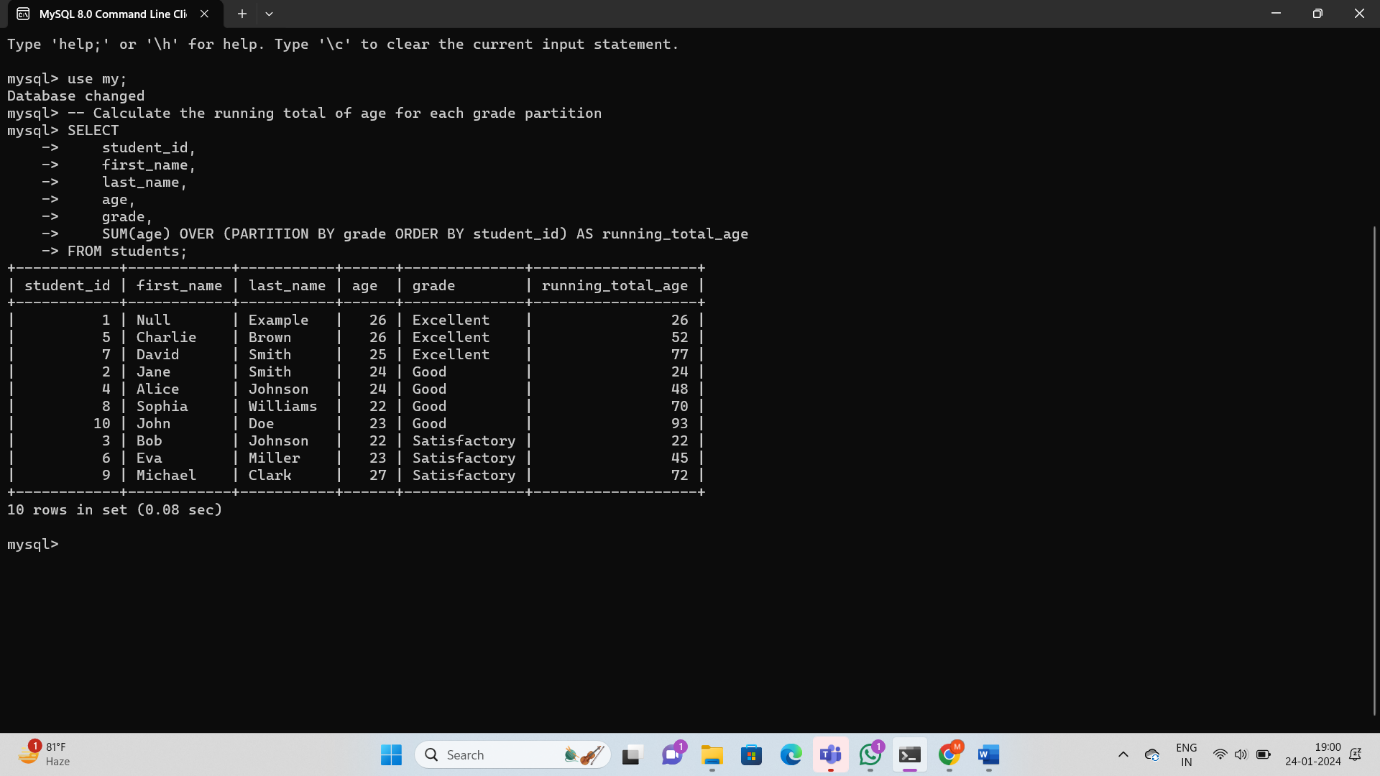
last\_name,

age,

grade,

SUM(age) OVER (PARTITION BY grade ORDER BY student\_id) AS running\_total\_age

FROM students;

****

#### **Example 2: Using** OVER **and** PARTITION BY **for Total Aggregation:**

-- Calculate the total age for each grade partition

SELECT

student\_id,

first\_name,

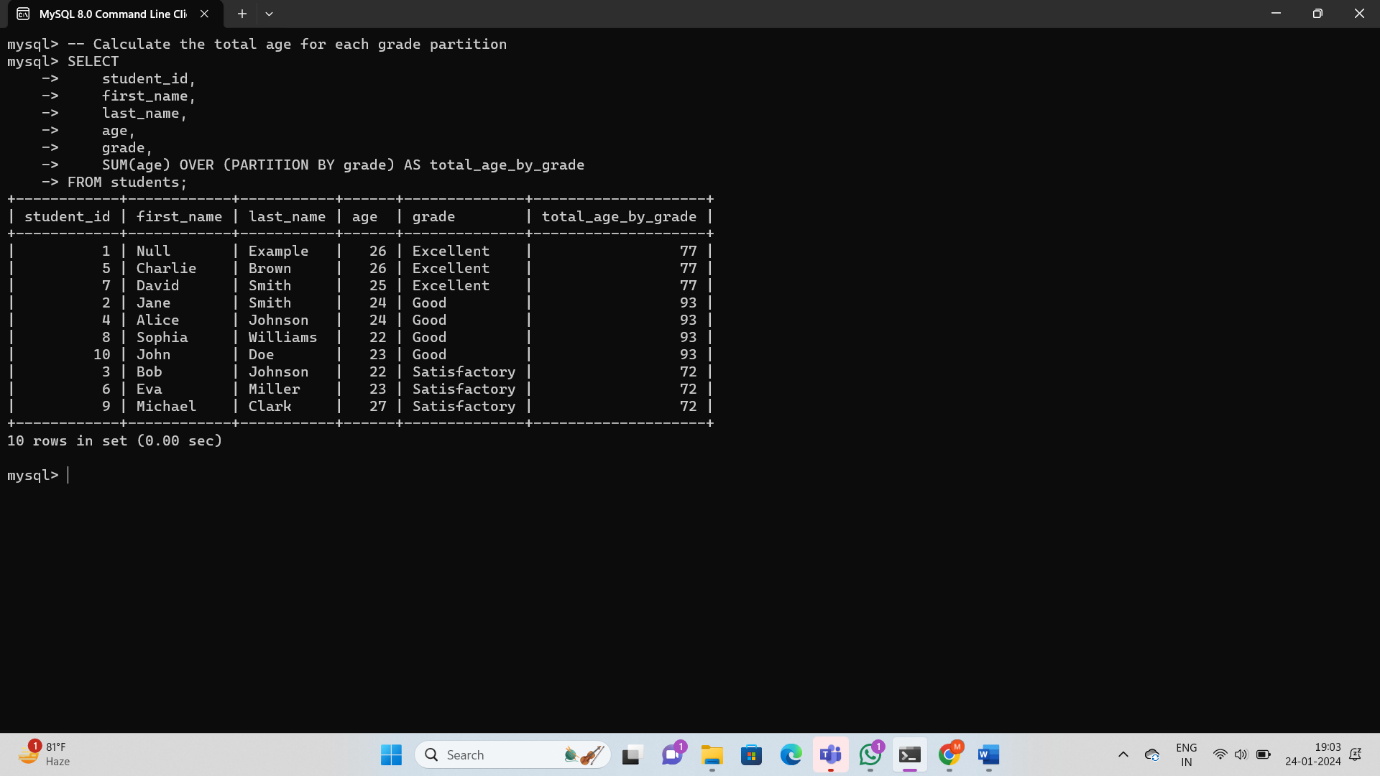
last\_name,

age,

grade,

SUM(age) OVER (PARTITION BY grade) AS total\_age\_by\_grade

FROM students;

****

#### **Example 3: Using** GROUP BY **to Calculate Subtotals:**

-- Calculate the subtotal of age for each grade

SELECT

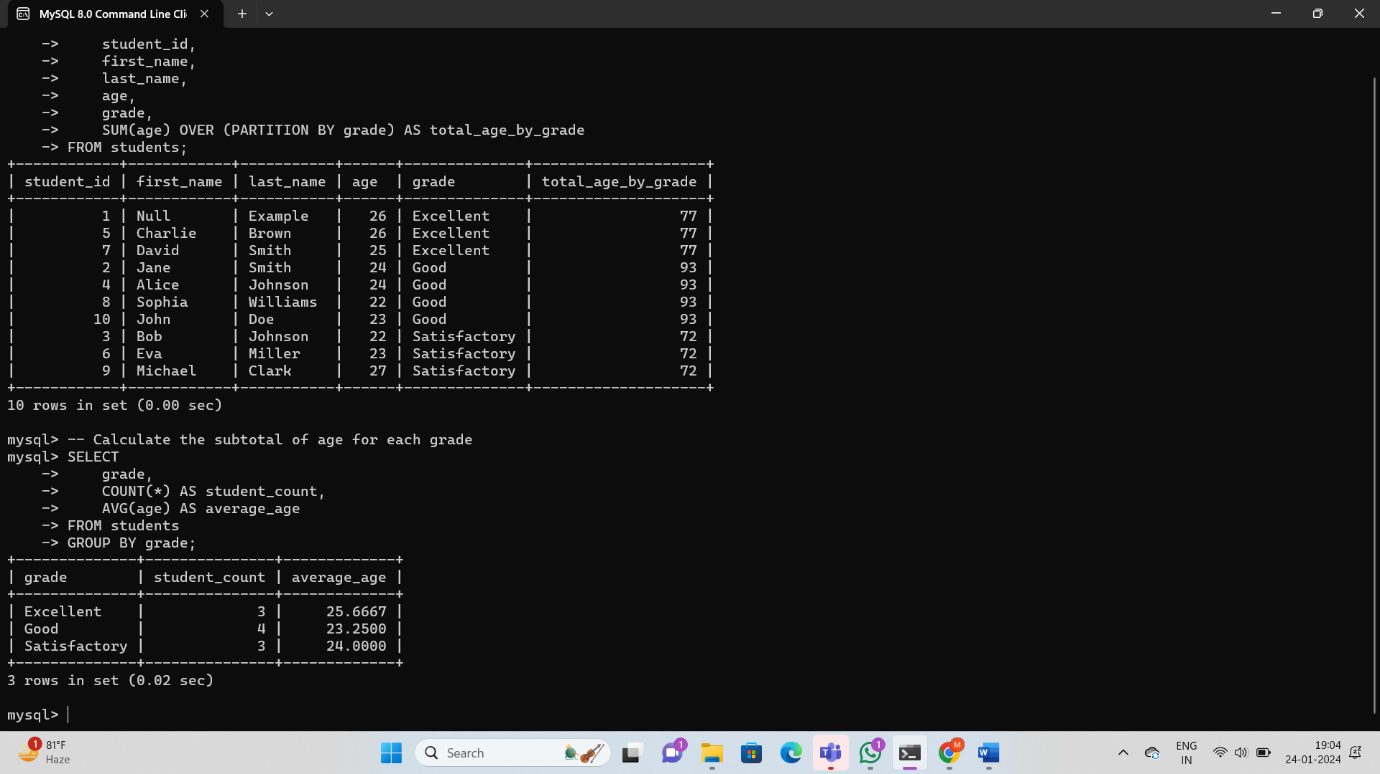
grade,

COUNT(\*) AS student\_count,

AVG(age) AS average\_age

FROM students

GROUP BY grade;

****

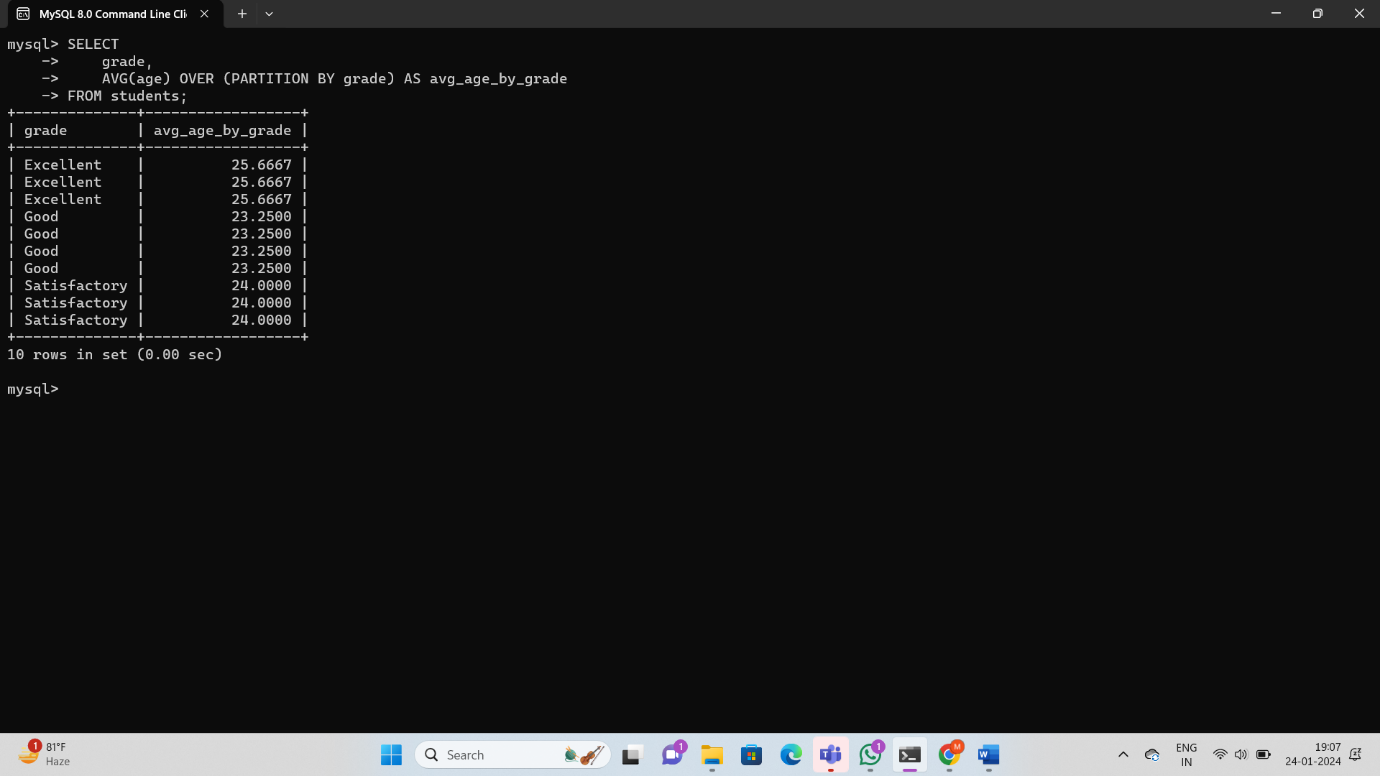
#### **Example: Calculate the average age for each grade using** OVER **and** PARTITION BY

SELECT

grade,

AVG(age) OVER (PARTITION BY grade) AS avg\_age\_by\_grade

FROM students;



### Total Aggregation using OVER and PARTITION BY:

#### **Example: Calculate the total age for each grade using** OVER **and** PARTITION BY**.**

SELECT

student\_id,

first\_name,

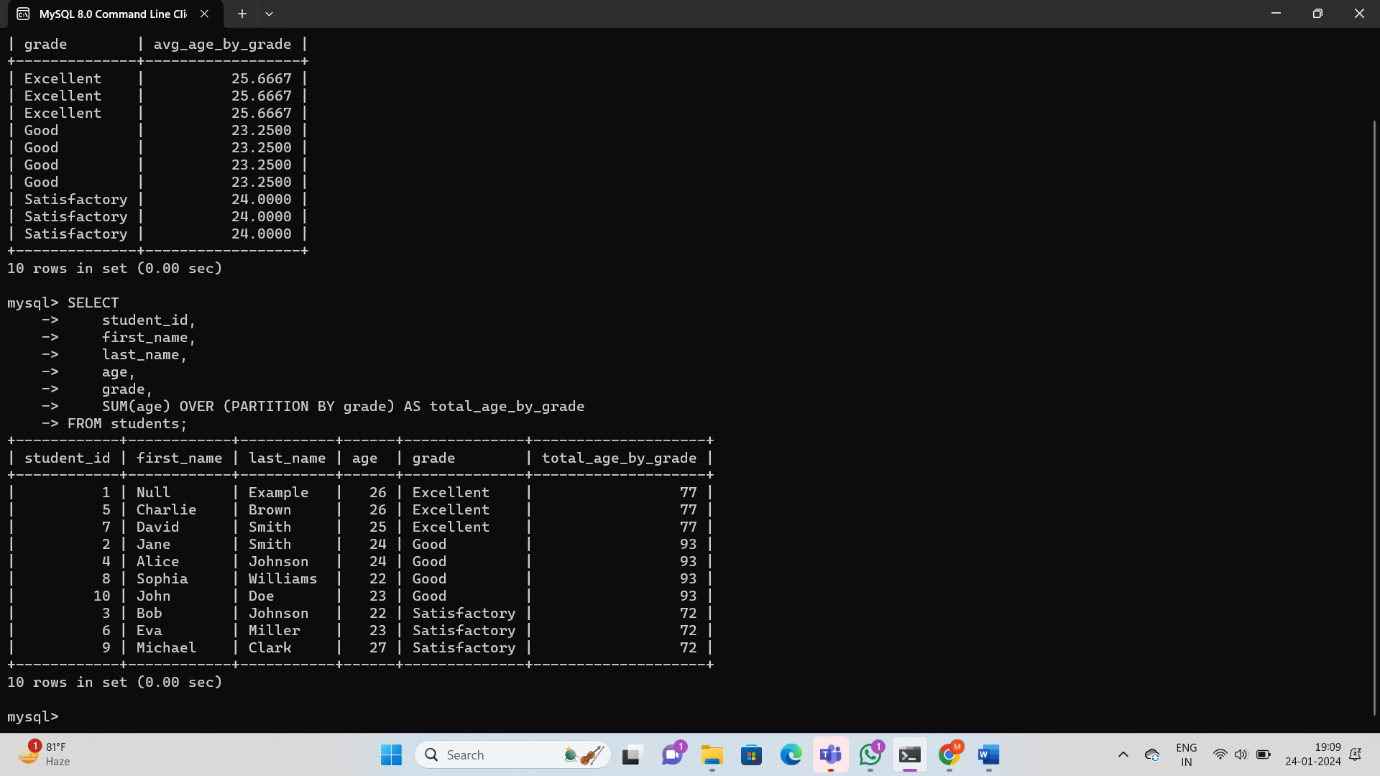
last\_name,

age,

grade,

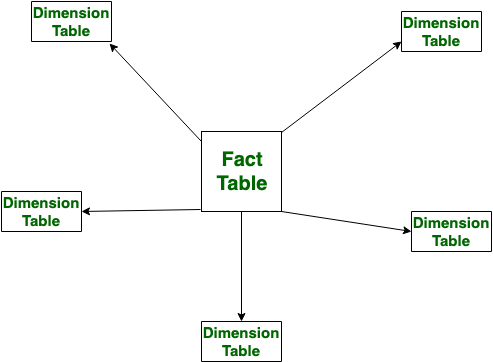
SUM(age) OVER (PARTITION BY grade) AS total\_age\_by\_grade

FROM students;



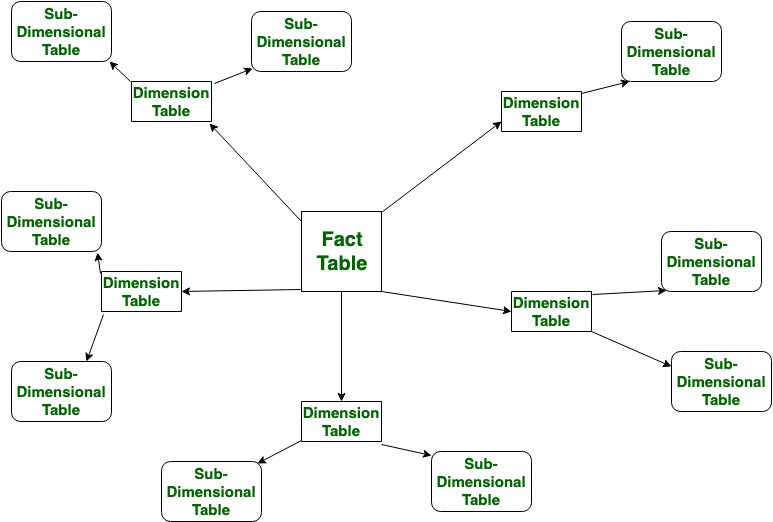
**Star Schema:**

 Star schema is the type of multidimensional model which is used for data warehouse. In star schema, the fact tables and the dimension tables are contained. In this schema fewer foreign-key join is used. This schema forms a star with fact table and dimension tables.



**Snowflake Schema:**

Snowflake Schema is also the type of multidimensional model which is used for [data warehouse](https://www.geeksforgeeks.org/data-warehousing/). In snowflake schema, the fact tables, dimension tables as well as sub dimension tables are contained. This schema forms a snowflake with fact tables, dimension tables as well as sub-dimension tables.



### **Star Schema Example:**

In a star schema, you typically have a central fact table surrounded by dimension tables.

#### **Fact Table:** enrolments

CREATE TABLE enrollments (

enrollment\_id INT PRIMARY KEY,

student\_id INT,

course\_id INT,

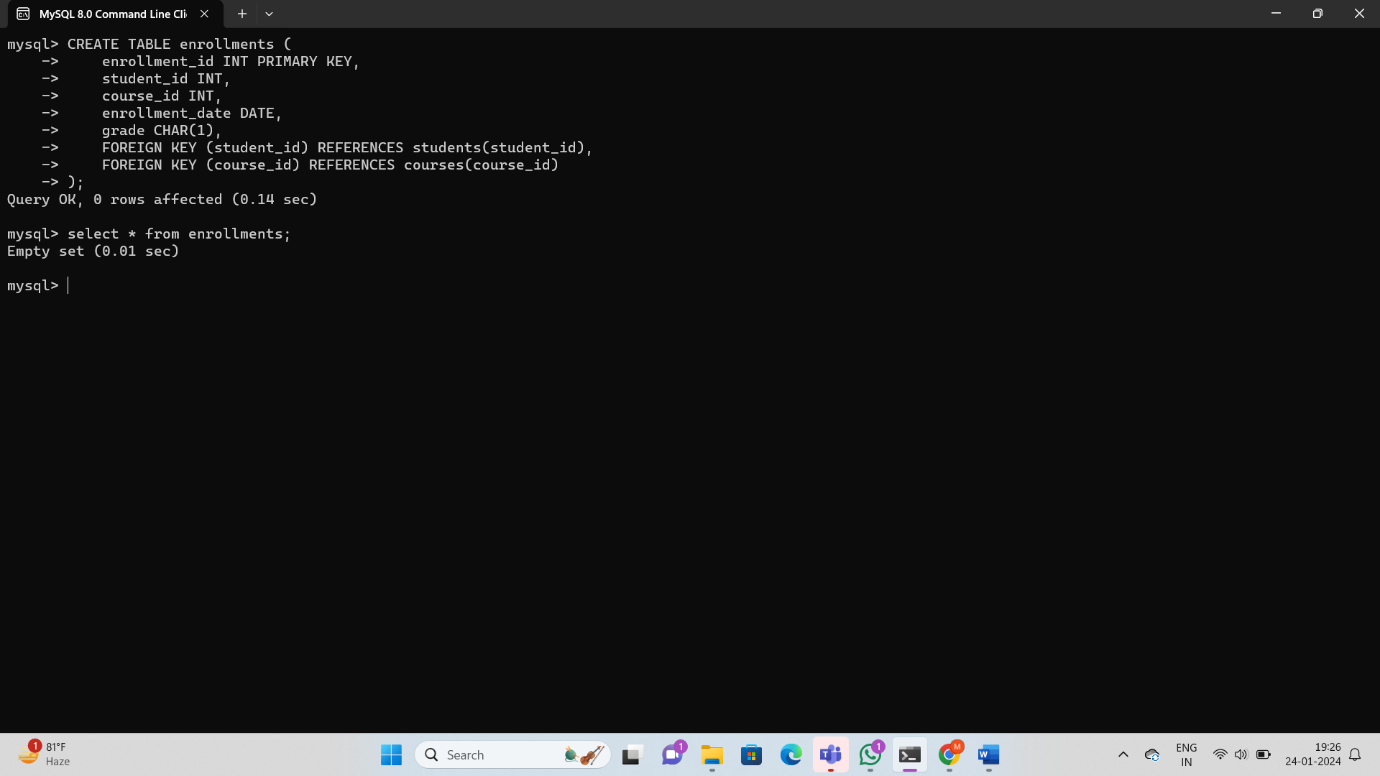
enrollment\_date DATE,

grade CHAR(1),

FOREIGN KEY (student\_id) REFERENCES students(student\_id),

FOREIGN KEY (course\_id) REFERENCES courses(course\_id)

);



#### **Dimension Tables:** students **and** courses

Your existing **students** and **courses** tables can be considered dimensions.

### Snowflaking Example:

Snowflaking involves normalizing dimension tables by breaking them into multiple related tables. In the context of the **students** table, let's say you have additional information about students stored in a separate table.

#### **Original** students **Table:**

CREATE TABLE students (

student\_id INT PRIMARY KEY,

first\_name VARCHAR(50),

last\_name VARCHAR(50),

age INT,

grade CHAR(1)

);

#### **Snowflaked** students **Table with Additional Information:**

CREATE TABLE students (

student\_id INT PRIMARY KEY,

first\_name VARCHAR(50),

last\_name VARCHAR(50),

age INT,

grade CHAR(1),

address\_id INT, -- Snowflaked into a separate table

FOREIGN KEY (address\_id) REFERENCES addresses(address\_id)

);

CREATE TABLE addresses (

address\_id INT PRIMARY KEY,

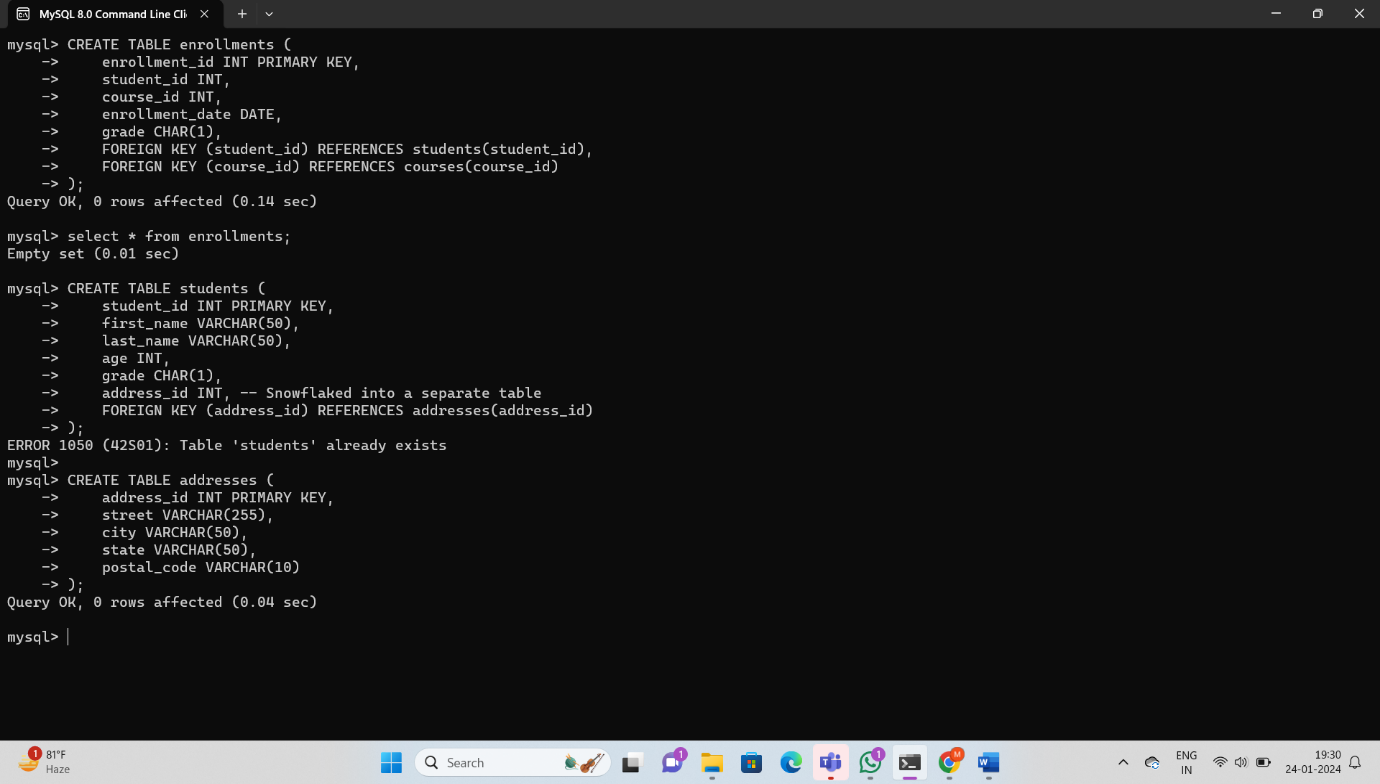
street VARCHAR(255),

city VARCHAR(50),

state VARCHAR(50),

postal\_code VARCHAR(10)

);



In this example, the **students** table is snowflaked by moving the address information into a separate table (**addresses**). The **address\_id** in the **students** table is a foreign key that references the **addresses** table.

### **Rules and Restrictions to Group and Filter Data:**

When using the **GROUP BY** clause in SQL, there are rules and restrictions that you should be aware of to ensure that your queries are correct and efficient. Here are some key rules and restrictions:

### **1. Aggregations in SELECT List:**

Columns in the **SELECT** list that are not part of an aggregate function must be included in the **GROUP BY** clause. This ensures that each non-aggregated column corresponds to a unique grouping.

-- Incorrect: 'name' is not part of an aggregate function or in the GROUP BY clause

SELECT name, COUNT(id)

FROM your\_table

GROUP BY id;

-- Correct: 'name' is in the GROUP BY clause

SELECT name, COUNT(id)

FROM your\_table

GROUP BY name;

### **2. Aggregate Functions in SELECT List:**

Columns in the **SELECT** list that are part of an aggregate function (e.g., **COUNT**, **SUM**, **AVG**) do not need to be included in the **GROUP BY** clause.

-- Correct: Using COUNT(id) as an aggregate function

SELECT name, COUNT(id)

FROM your\_table

GROUP BY name;

### **3. HAVING Clause for Filtering Grouped Data:**

Filtering on aggregated data should be done using the **HAVING** clause, not the **WHERE** clause.

-- Incorrect: Using WHERE for aggregated filtering

SELECT name, COUNT(id)

FROM your\_table

WHERE COUNT(id) > 1

GROUP BY name;

-- Correct: Using HAVING for aggregated filtering

SELECT name, COUNT(id)

FROM your\_table

GROUP BY name

HAVING COUNT(id) > 1;

### **4. No Aliases in WHERE or HAVING:**

You cannot use column aliases defined in the **SELECT** list directly in the **WHERE** or **HAVING** clauses. This is because the **WHERE** and **HAVING** clauses are processed before the **SELECT** list.

-- Incorrect: Using alias 'total' in the WHERE clause

SELECT name, COUNT(id) AS total

FROM your\_table

WHERE total > 1

GROUP BY name;

-- Correct: Repeating the aggregate function in the WHERE clause

SELECT name, COUNT(id) AS total

FROM your\_table

GROUP BY name

HAVING COUNT(id) > 1;

### **5. GROUP BY on Selected Columns:**

When using **GROUP BY**, all selected columns that are not part of an aggregate function must be included in the **GROUP BY** clause.

-- Incorrect: 'age' is not in the GROUP BY clause

SELECT name, age, COUNT(id)

FROM your\_table

GROUP BY name;

-- Correct: Including 'age' in the GROUP BY clause

SELECT name, age, COUNT(id)

FROM your\_table

### GROUP BY name, age;

### **6. ORDER BY in Aggregated Queries:**

When using **GROUP BY**, the **ORDER BY** clause should include the columns used in the **GROUP BY** clause or the position of the columns in the **SELECT** list.

-- Correct: Ordering by 'name' which is part of GROUP BY

SELECT name, COUNT(id)

FROM your\_table

GROUP BY name

ORDER BY name;

-- Correct: Ordering by the second column in the SELECT list

SELECT name, COUNT(id)

FROM your\_table

GROUP BY name

ORDER BY 2;

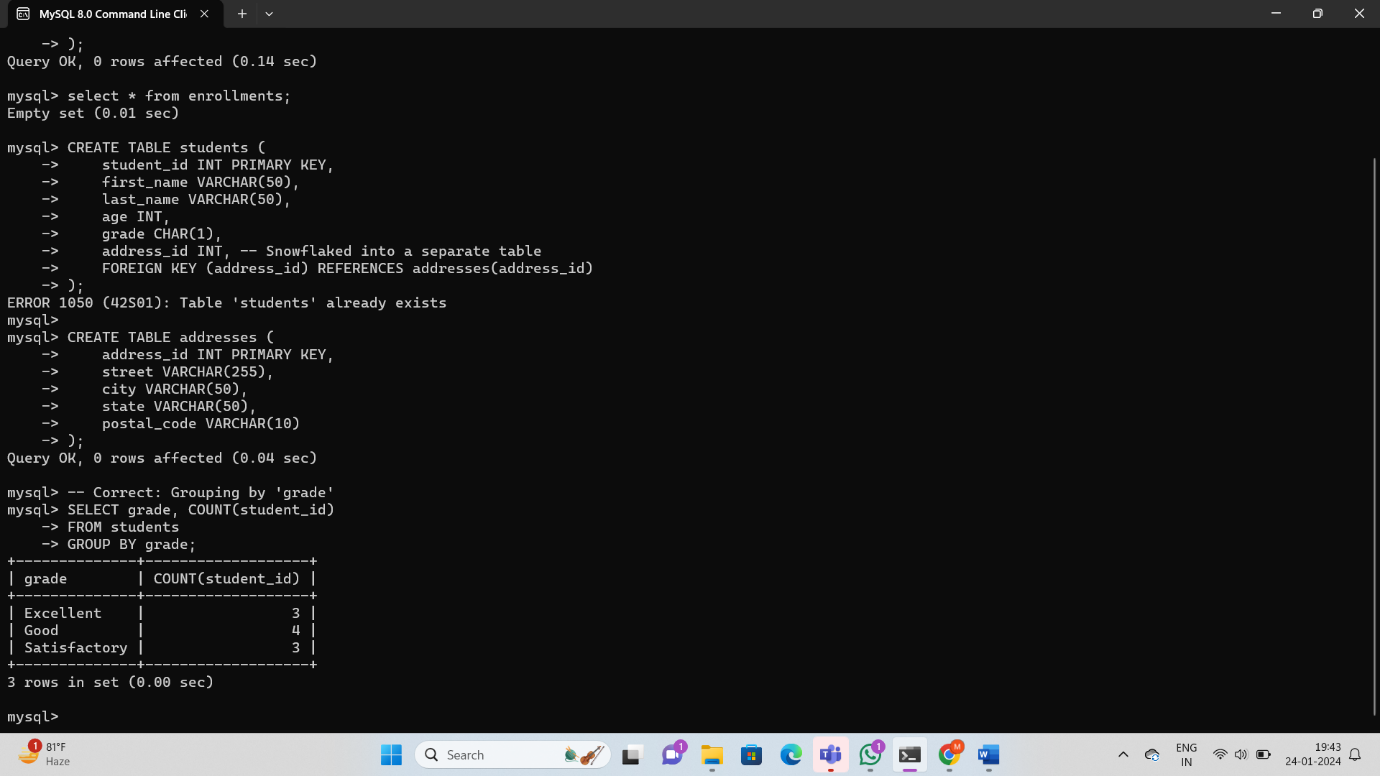
#### **Example 1: Correct GROUP BY:**

-- Correct: Grouping by 'grade'

SELECT grade, COUNT(student\_id)

FROM students

GROUP BY grade;



#### **Example 2: Incorrect GROUP BY:**

-- Incorrect: 'name' is not part of GROUP BY or an aggregate function

SELECT first\_name, COUNT(student\_id)

FROM students

GROUP BY grade;

#### **Example 3: Using HAVING for Aggregated Filtering:**

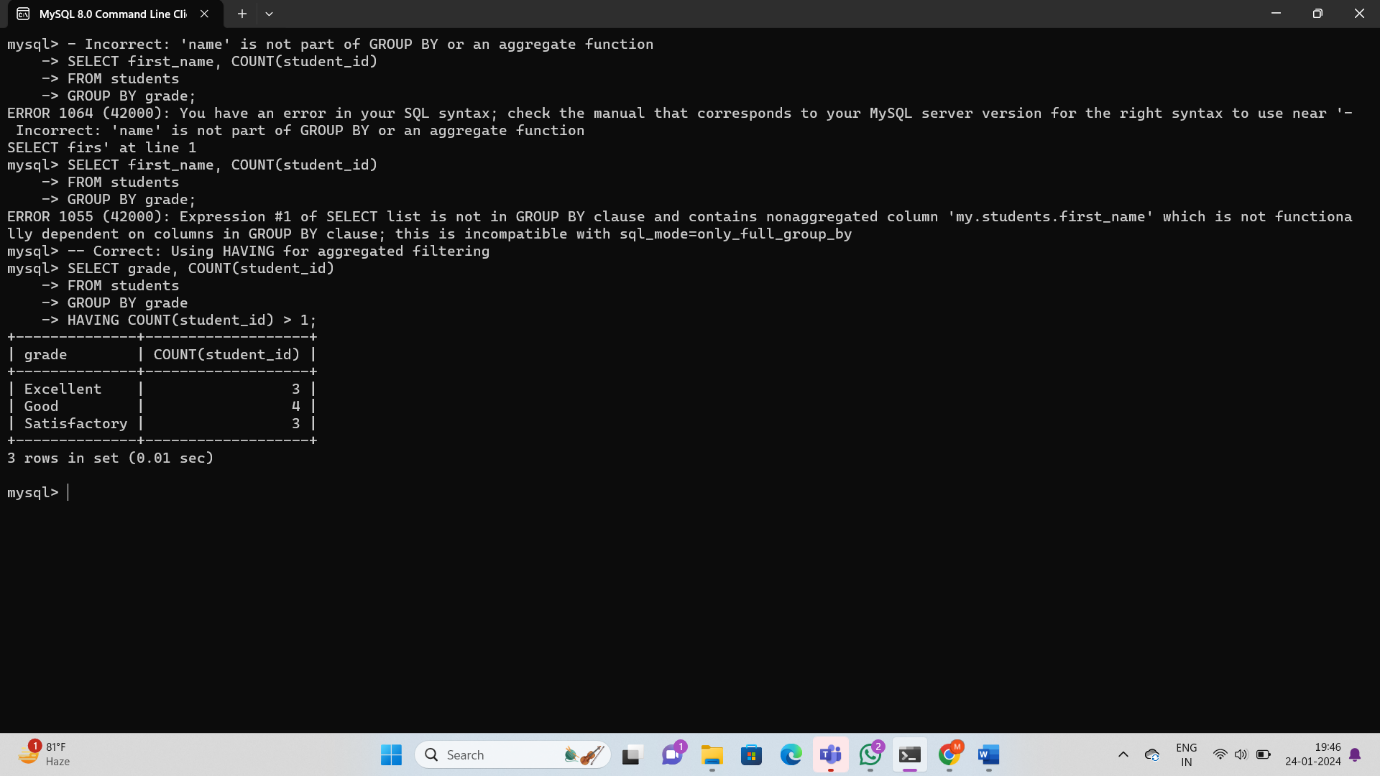
-- Correct: Using HAVING for aggregated filtering

SELECT grade, COUNT(student\_id)

FROM students

GROUP BY grade

HAVING COUNT(student\_id) > 1;



#### **Example 4: GROUP BY with JOIN:**

-- Correct: GROUP BY with JOIN

SELECT c.course\_id, c.course\_name, COUNT(s.student\_id) AS student\_count

FROM courses c

JOIN students s ON c.student\_id = s.student\_id

GROUP BY c.course\_id, c.course\_name;

#### 

#### **Example 5: ORDER BY with GROUP BY:**

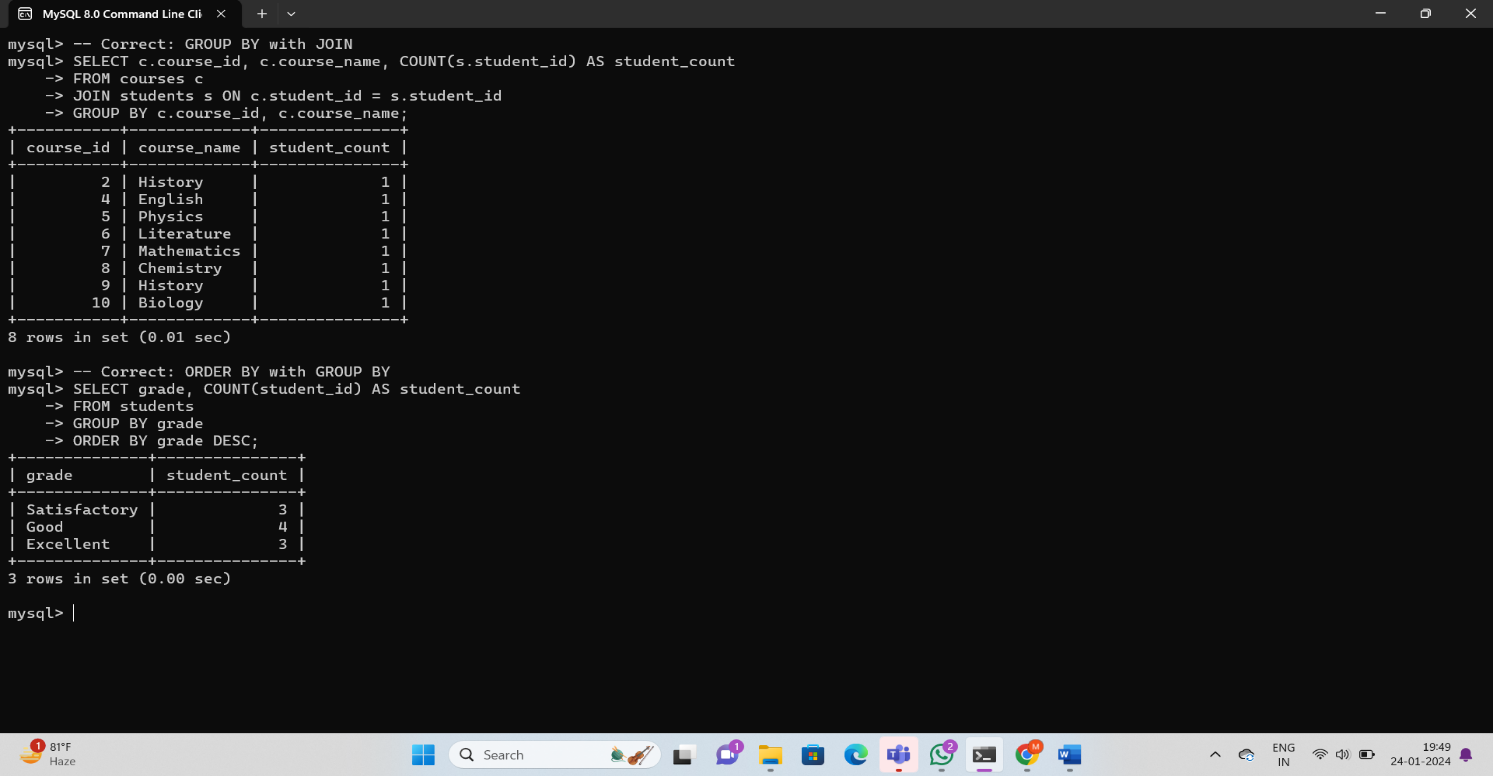
-- Correct: ORDER BY with GROUP BY

SELECT grade, COUNT(student\_id) AS student\_count

FROM students

GROUP BY grade

ORDER BY grade DESC;



These examples demonstrate correct and incorrect usage of the **GROUP BY** clause, **HAVING** clause for filtering, and the **ORDER BY** clause with **GROUP BY**. Always ensure that the columns in the **SELECT** list are either part of an aggregate function or included in the **GROUP BY** clause. Use the **HAVING** clause for filtering aggregated data, and order the results appropriately with the **ORDER BY** clause.

### **Order of Execution of SQL Queries:**

The order of execution for SQL queries generally follows these steps:

1. **FROM clause:** Specifies the tables from which to retrieve data.
2. **WHERE clause:** Filters rows based on specified conditions.
3. **GROUP BY clause:** Groups rows based on specified columns.
4. **HAVING clause:** Filters grouped rows based on specified conditions.
5. **SELECT clause:** Specifies columns to be included in the result set.
6. **ORDER BY clause:** Sorts the result set based on specified columns.

#### **Example:**

SELECT grade, AVG(age) AS avg\_age

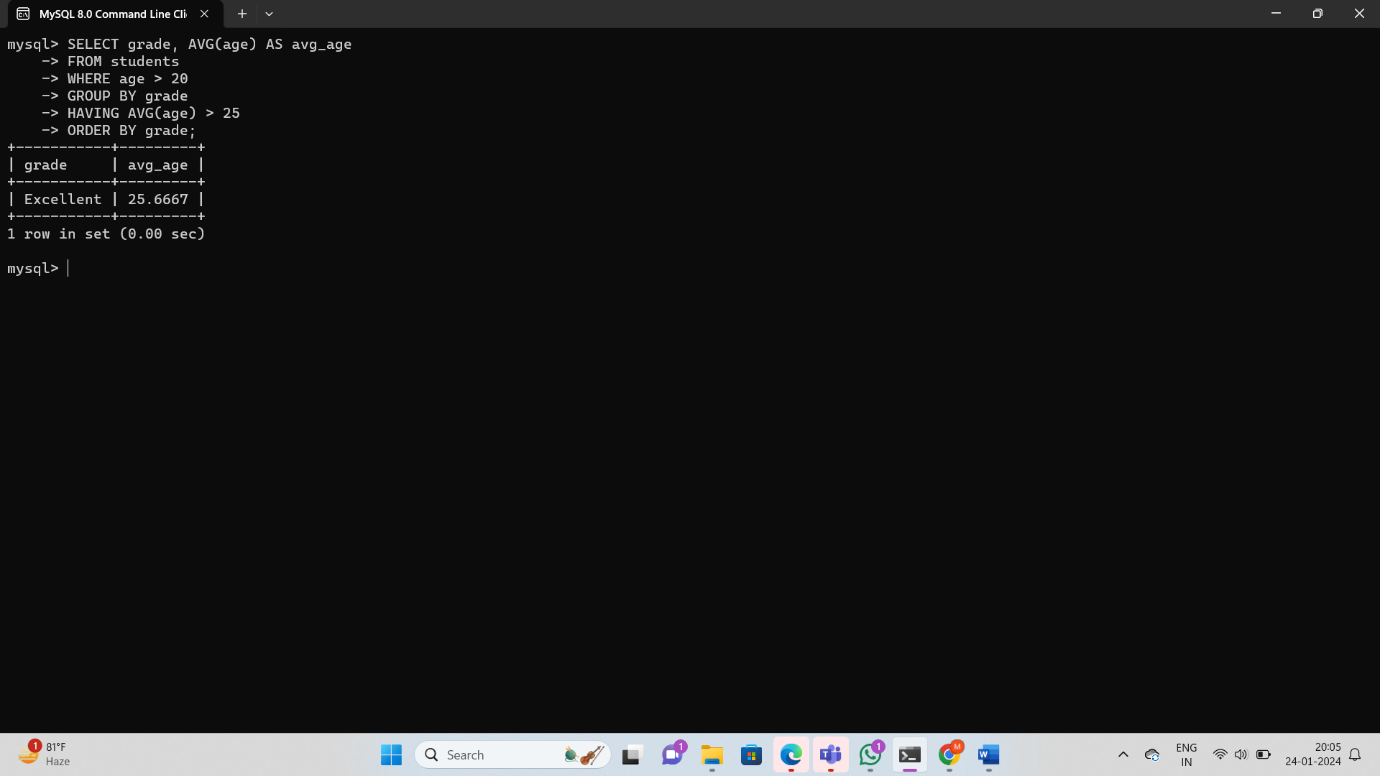
FROM students

WHERE age > 20

GROUP BY grade

HAVING AVG(age) > 25

ORDER BY grade;



### **How to Calculate Subtotals in SQL Queries:**

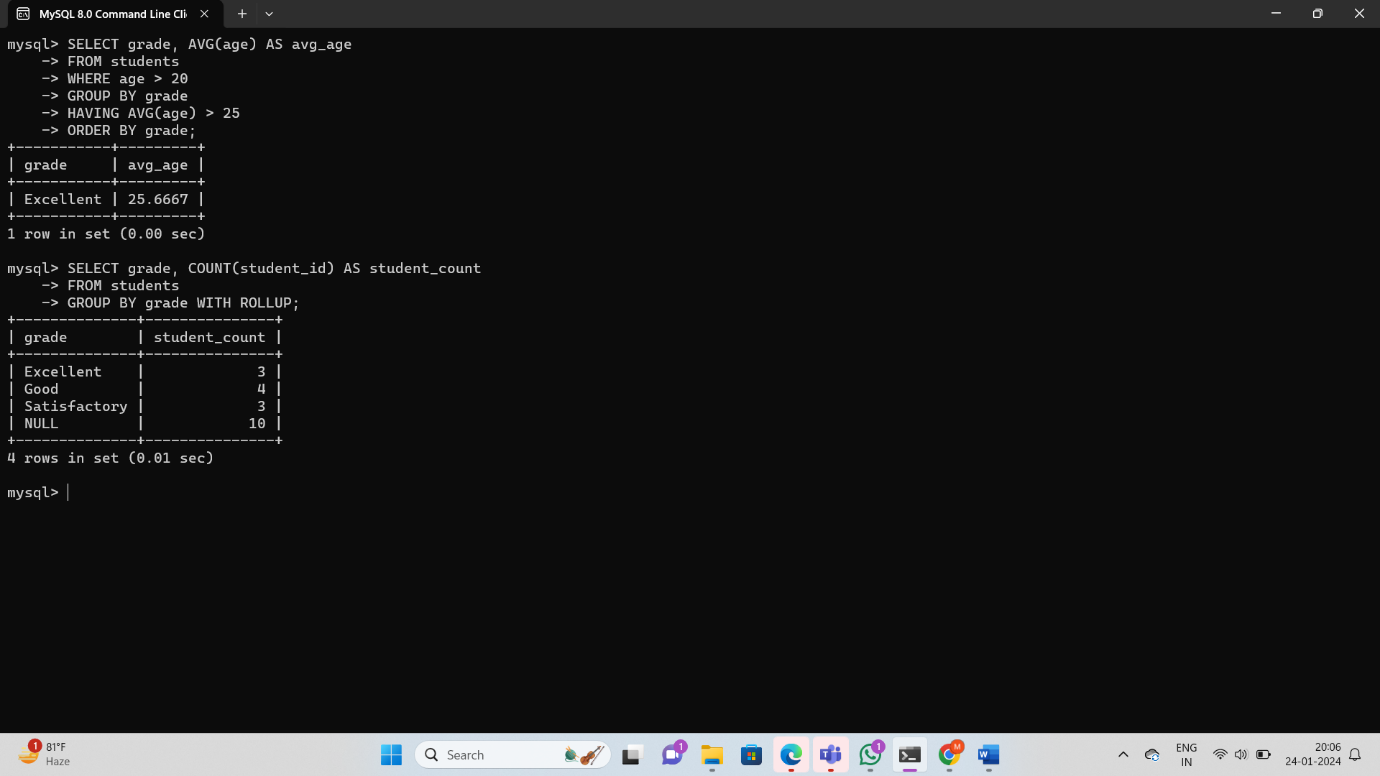
Calculating subtotals often involves using the **GROUP BY** clause along with the **WITH ROLLUP** option. The **WITH ROLLUP** option produces subtotals and a grand total for the grouped data.

#### **Example:**

SELECT grade, COUNT(student\_id) AS student\_count

FROM students

GROUP BY grade WITH ROLLUP;



### **Differences Between UNION, EXCEPT, and INTERSECT Operators:**

#### **UNION:**

The **UNION** operator combines the result sets of two or more SELECT statements into a single result set, removing duplicate rows.

#### **EXCEPT:**

The **EXCEPT** operator returns the distinct rows that appear in the result of the first SELECT statement but not in the result of the second SELECT statement.

#### **INTERSECT:**

The **INTERSECT** operator returns the distinct rows that appear in both the result of the first SELECT statement and the result of the second SELECT statement.

#### **Examples:**

-- UNION: Combine student names from 'students' and 'courses'

SELECT first\_name AS name FROM students

UNION

SELECT course\_name AS name FROM courses;

-- EXCEPT: Find students who are not enrolled in any course

SELECT first\_name, last\_name FROM students

EXCEPT

SELECT s.first\_name, s.last\_name FROM students s

JOIN courses c ON s.student\_id = c.student\_id;

-- INTERSECT: Find students who are enrolled in courses

SELECT first\_name, last\_name FROM students

INTERSECT

SELECT s.first\_name, s.last\_name FROM students s

JOIN courses c ON s.student\_id = c.student\_id;

#### **UNION:**

The SQL UNION Operator

The UNION operator is used to combine the result-set of two or more SELECT statements.

* Every SELECT statement within UNION must have the same number of columns
* The columns must also have similar data types
* The columns in every SELECT statement must also be in the same order

### **UNION Syntax**

SELECT column\_name(s) FROM table1  
UNION  
SELECT column\_name(s) FROM table2;

### **UNION ALL Syntax**

The UNION operator selects only distinct values by default. To allow duplicate values, use UNION ALL:

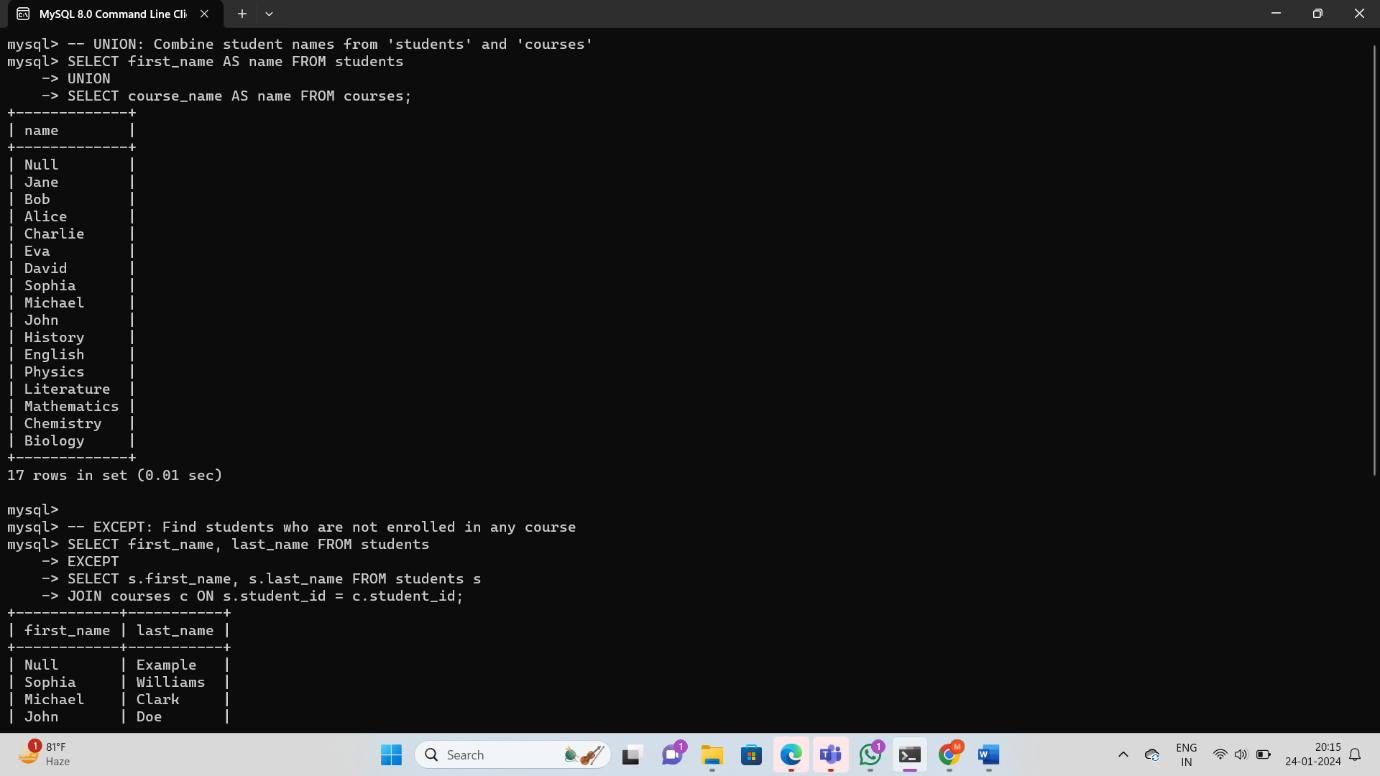
SELECT column\_name(s) FROM table1  
UNION ALL  
SELECT column\_name(s) FROM table2;

-- UNION: Combine student names from 'students' and 'courses'

SELECT first\_name AS name FROM students

UNION

SELECT course\_name AS name FROM courses;



#### **EXCEPT:**

EXCEPT returns distinct rows from the left input query that aren't output by the right input query.

In SQL, EXCEPT returns those tuples that are returned by the first SELECT operation, and not returned by the second SELECT operation.

### **Syntax**

{ <query\_specification> | ( <query\_expression> ) }

{ EXCEPT | INTERSECT }

{ <query\_specification> | ( <query\_expression> ) }

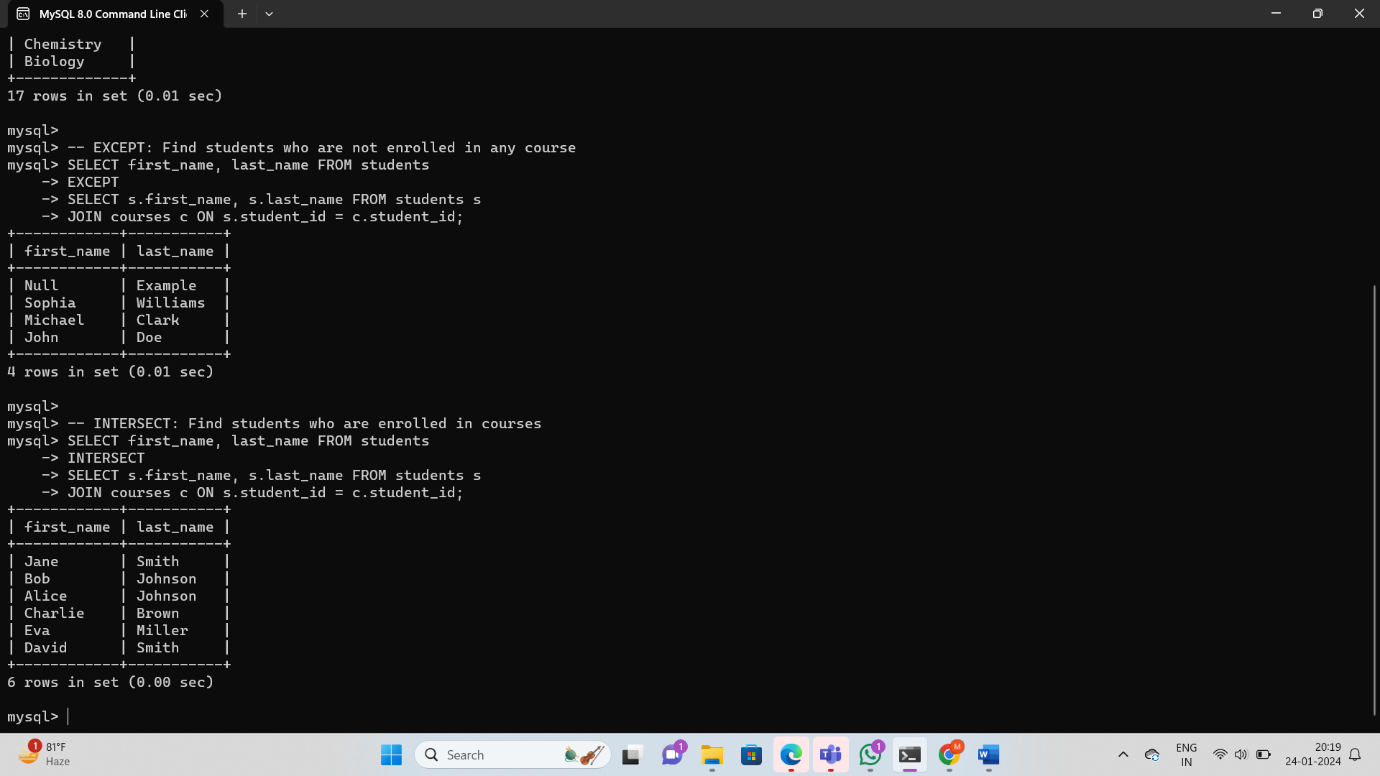
-- EXCEPT: Find students who are not enrolled in any course

SELECT first\_name, last\_name FROM students

EXCEPT

SELECT s.first\_name, s.last\_name FROM students s

JOIN courses c ON s.student\_id = c.student\_id;



#### **INTERSECT:**

The INTERSECT clause in SQL is used to combine two [SELECT](https://www.geeksforgeeks.org/sql-select-clause/) statements but the dataset returned by the INTERSECT statement will be the intersection of the data sets of the two SELECT statements. In simple words, the INTERSECT statement will return only those rows which will be common to both of the SELECT statements.

### **Syntax**

SELECT column1, column2 ….

FROM table\_names

WHERE condition

**INTERSECT**

SELECT column1, column2 ….

FROM table\_names

WHERE condition

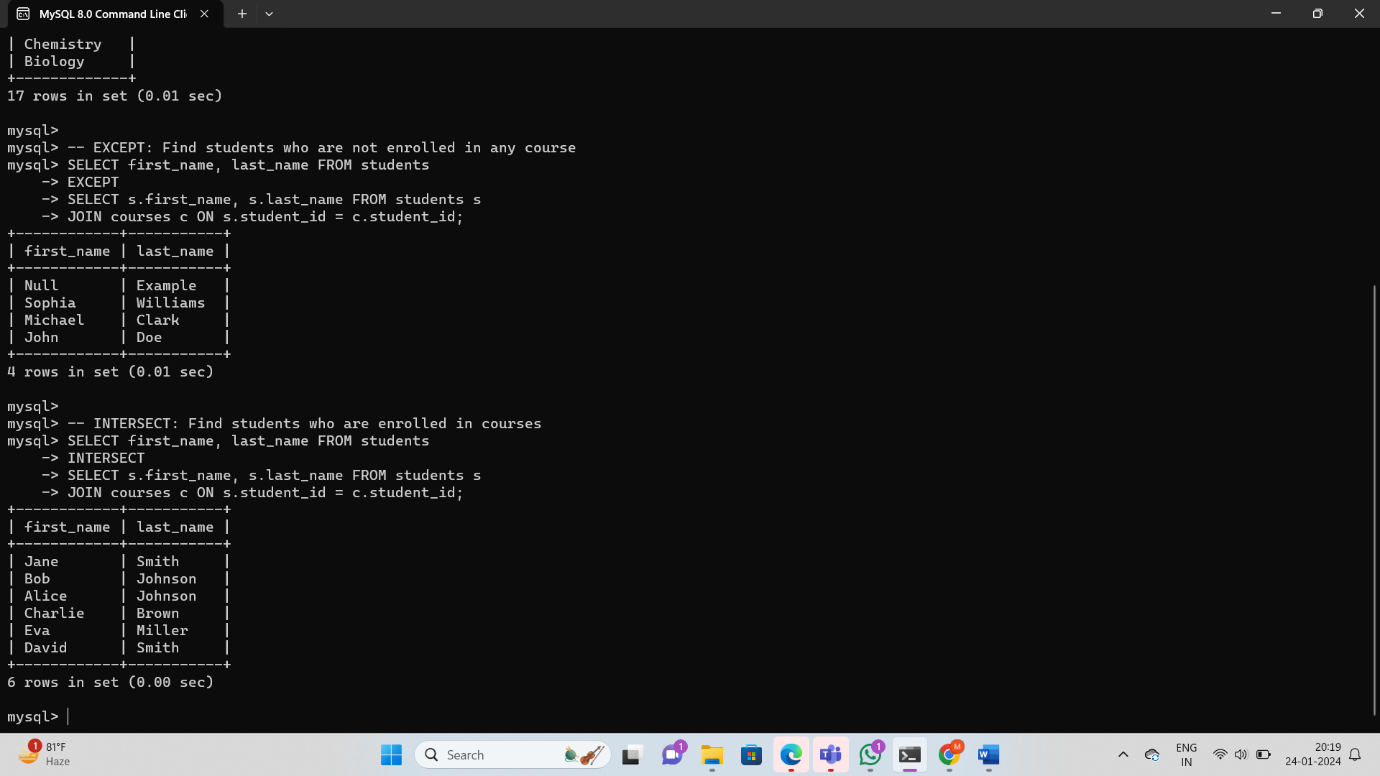
-- INTERSECT: Find students who are enrolled in courses

SELECT first\_name, last\_name FROM students

INTERSECT

SELECT s.first\_name, s.last\_name FROM students s

JOIN courses c ON s.student\_id = c.student\_id;

**

These examples demonstrate the usage of the **UNION**, **EXCEPT**, and **INTERSECT** operators in SQL queries. Adjust the queries based on your specific data and requirements.

### **Regular Expressions (REGEX) in SQL:**

Regular expressions allow you to perform pattern-matching operations on text. Some databases support regular expressions in SQL queries through functions like **REGEXP** or **RLIKE**. However, the syntax can vary between databases.

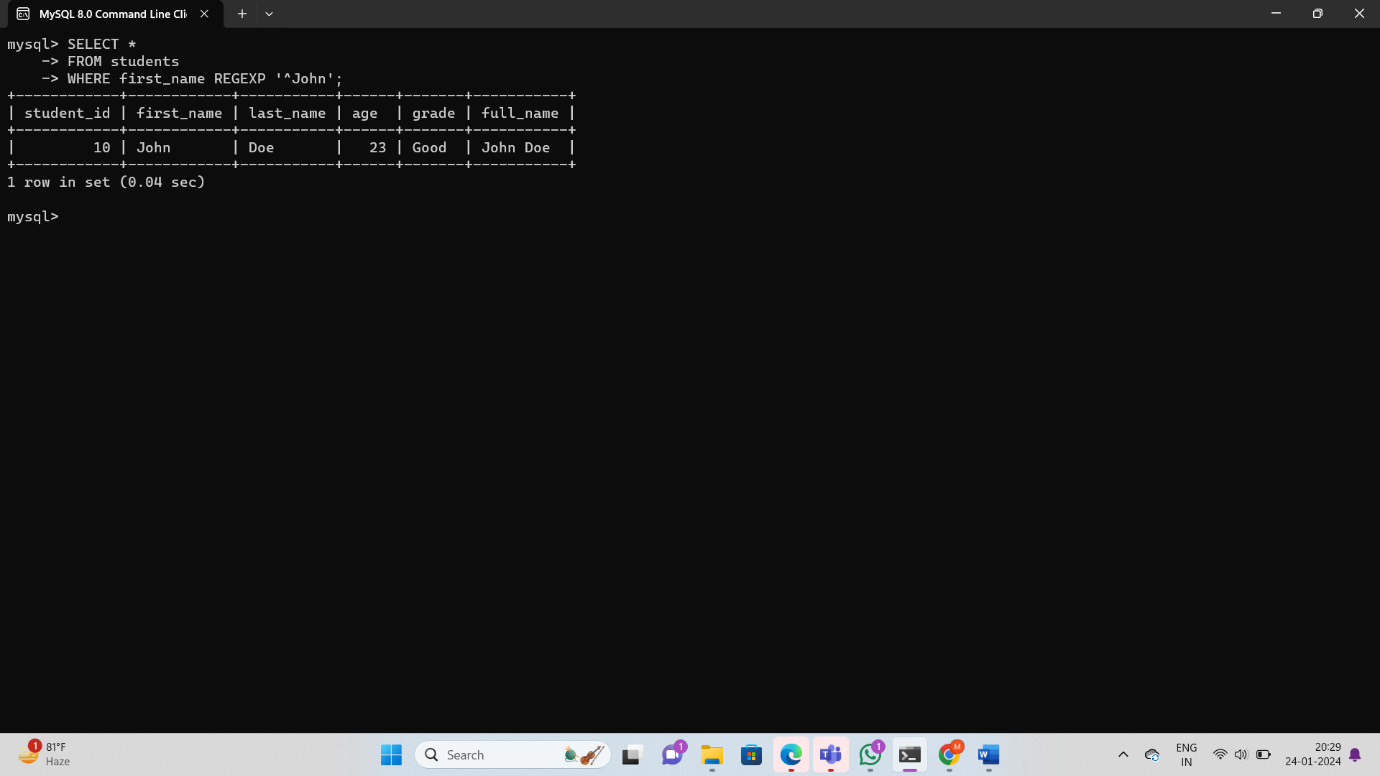
#### **Example (MySQL):**

Assuming you want to find names that start with "John" in a **students** table:

SELECT \*

FROM students

WHERE first\_name REGEXP '^John';



In this example, **^** represents the start of the string, and **John** is the pattern we're looking for.

### **Materialized Views:**

A materialized view is a database object that stores the results of a query and can be indexed or referenced like a table. Unlike a regular view, a materialized view physically stores the data, which can improve query performance at the cost of potential staleness.

-- Create a materialized view that stores the average age per grade

CREATE MATERIALIZED VIEW avg\_age\_per\_grade AS

SELECT grade, AVG(age) AS avg\_age

FROM students

GROUP BY grade;

-- Query the materialized view

SELECT \* FROM avg\_age\_per\_grade;