

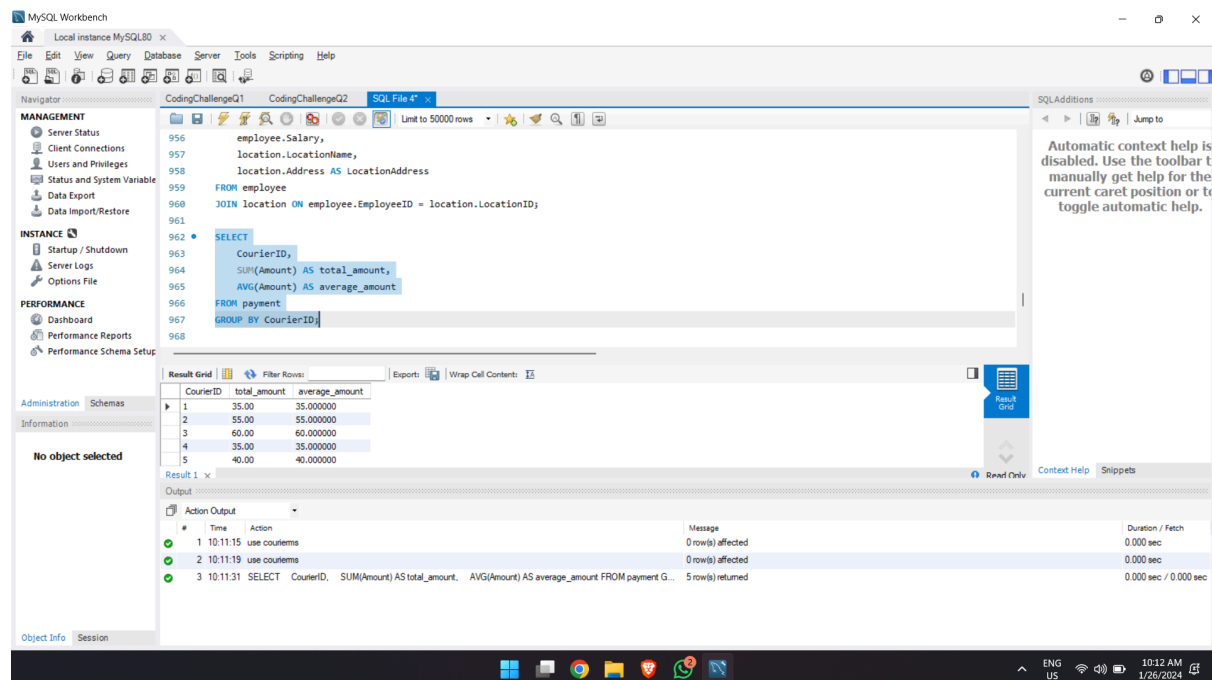
Name : Parth Nandedkar
Date : 24 Jan 2024
Topics : Advanced SQL Concepts
Batch : Data Engineering Batch-1

Rules and Restrictions to Group and Filter Data in SQL queries :

GROUP BY Clause:

When using aggregate functions (e.g., SUM, AVG, COUNT), the GROUP BY clause is required.

Every column in the SELECT list that is not part of an aggregate function must be included in the GROUP BY clause.



No Aggregates in SELECT Without GROUP BY:

If you use an aggregate function in the SELECT list, all non-aggregated columns must be included in the GROUP BY clause.

No Ambiguity:

Avoid ambiguous column names. If a column exists in multiple tables, specify the table alias or use table.column notation.

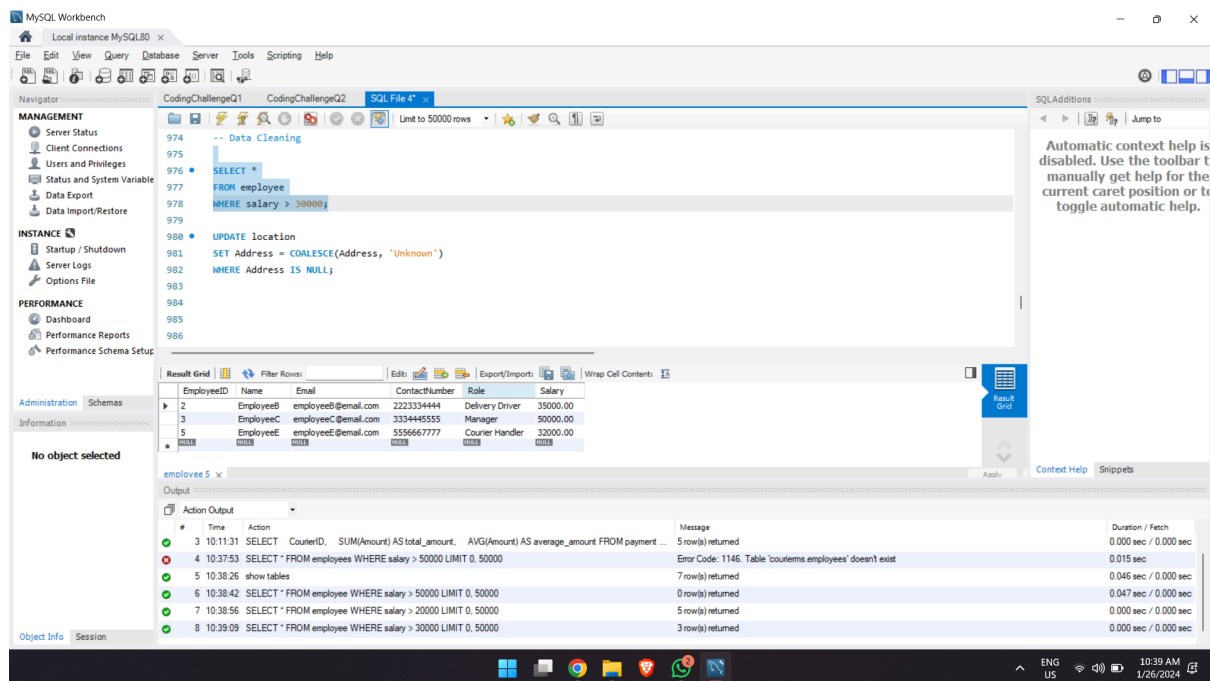
Filtering Data:

WHERE Clause for Row Filtering:

Use the WHERE clause to filter rows based on specific conditions.

Avoid Aggregates in WHERE:

Avoid using aggregate functions directly in the WHERE clause. Instead, use the HAVING clause for filtering aggregated results.

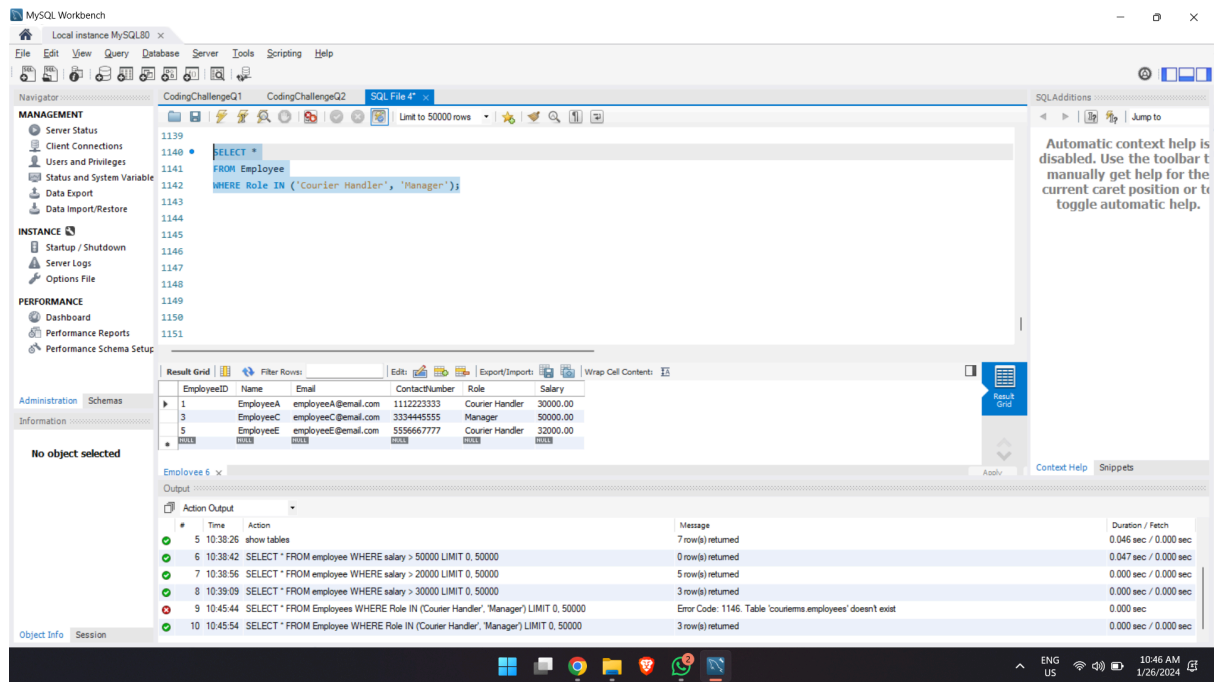


Logical Operators:

Use logical operators (AND, OR, NOT) to combine multiple conditions in the WHERE clause.

IN and NOT IN:

Use IN to filter rows where a column's value matches any value in a specified list. Use NOT IN for the opposite.



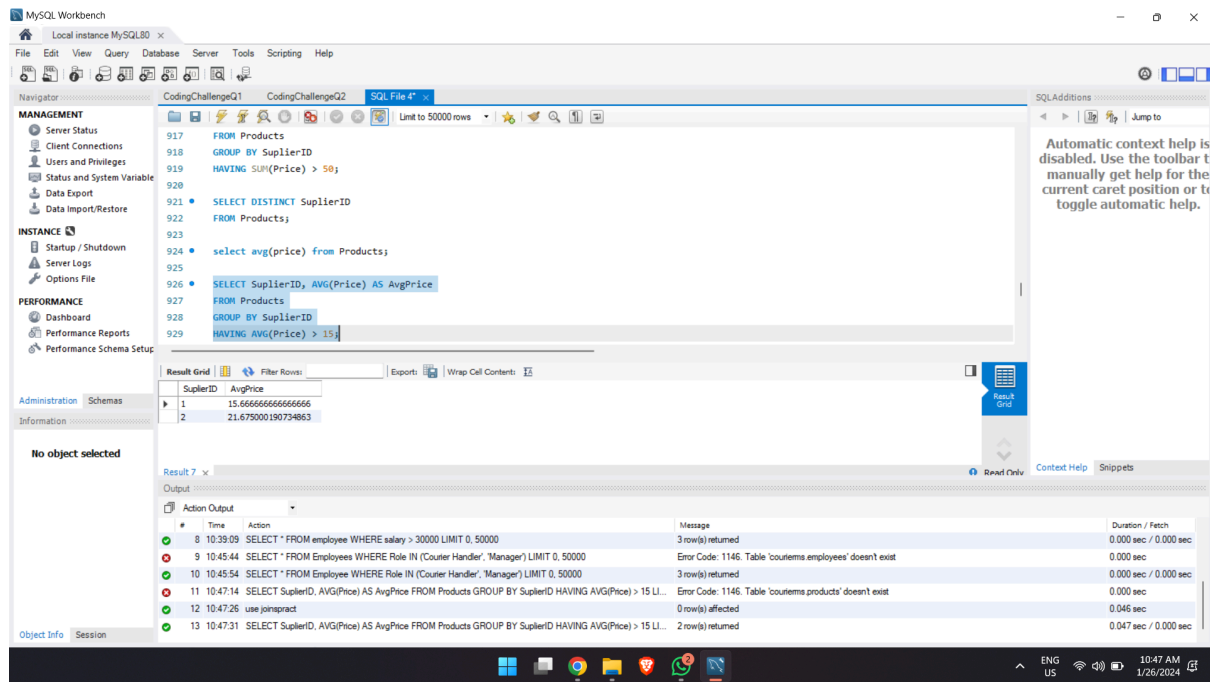
HAVING Clause:

Filtering Aggregated Results:

Use the HAVING clause to filter results based on aggregated values. It is similar to the WHERE clause but applies to grouped and aggregated data.

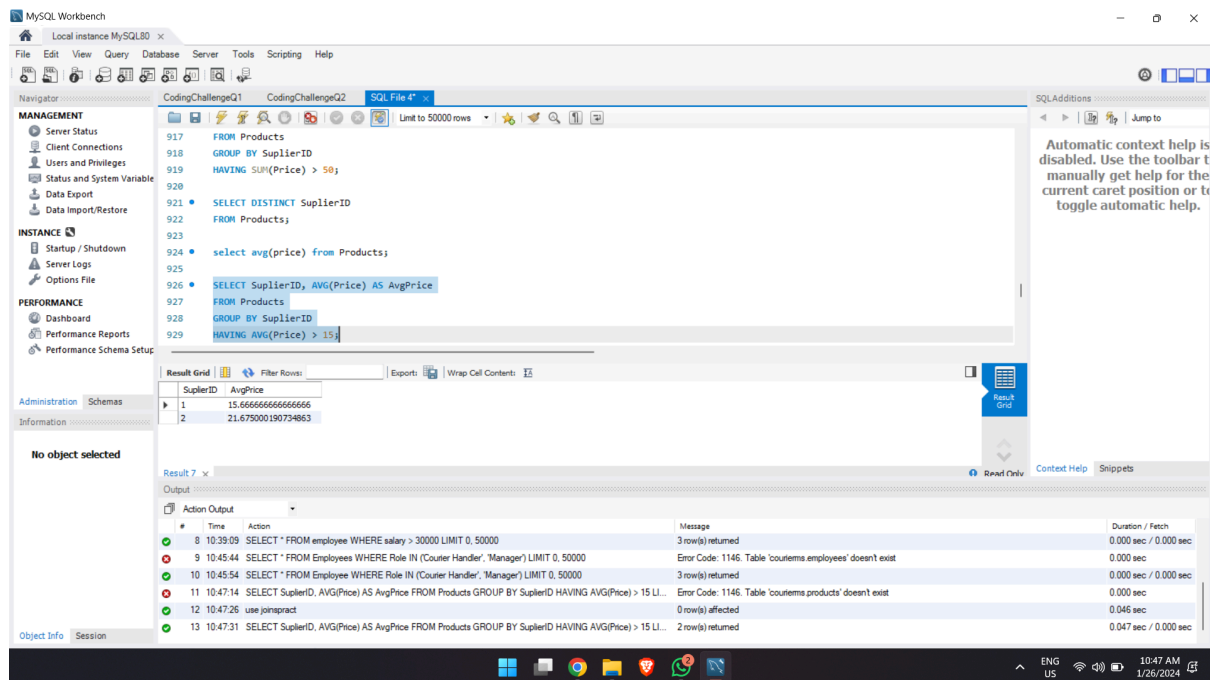
Aggregate Functions in HAVING:

You can use aggregate functions (e.g., SUM, AVG, COUNT) directly in the HAVING clause.



ORDER BY Clause: ORDER BY with GROUP BY:

If you are using GROUP BY, columns in the ORDER BY clause must be either part of the GROUP BY or used with an aggregate function.



Subqueries:

Subqueries for Complex Conditions:

For complex conditions, use subqueries to filter or compare results.

Correlated Subqueries:

In correlated subqueries, the inner query depends on the outer query. Be mindful of performance implications.

The screenshot shows the MySQL Workbench interface. The SQL editor contains the following query:

```
SELECT
  CourierID,
  SenderName,
  ReceiverName,
  (SELECT Amount FROM Payment WHERE CourierID = Courier.CourierID) AS PaymentAmount
FROM
  Courier
WHERE
  Status = 'Delivered';
```

The query is executed, and the results are displayed in the Result Grid:

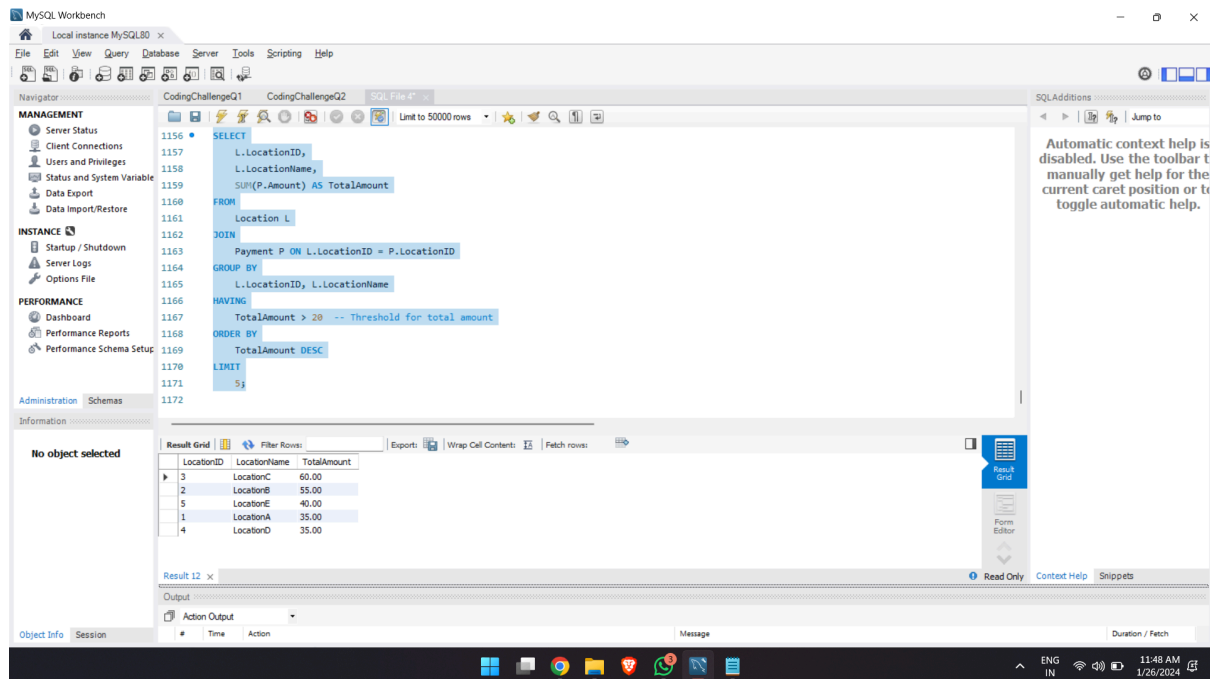
CourierID	SenderName	ReceiverName	PaymentAmount
2	SenderB	ReceiverB	55.00
5	SenderE	ReceiverE	40.00

The Output pane shows the execution log:

#	Time	Action	Message	Duration / Fetch
16	11:38:08	SELECT	Error Code: 1146. Table 'courierms.couriers' doesn't exist	0.000 sec
17	11:38:24	SELECT	Error Code: 1146. Table 'courierms.payments' doesn't exist	0.000 sec
18	11:38:37	show tables	7 row(s) returned	0.000 sec / 0.000 sec
19	11:39:05	SELECT	Error Code: 1054. Unknown column 'Couriers.CourierID' in 'where clause'	0.000 sec
20	11:40:12	select * from courier LIMIT 0, 50000	5 row(s) returned	0.000 sec / 0.000 sec
21	11:40:50	SELECT	2 row(s) returned	0.000 sec / 0.000 sec

Order of Execution of SQL Queries :

In what order the query is going to executes helps to understand, I have Executed the query and explained execution order.

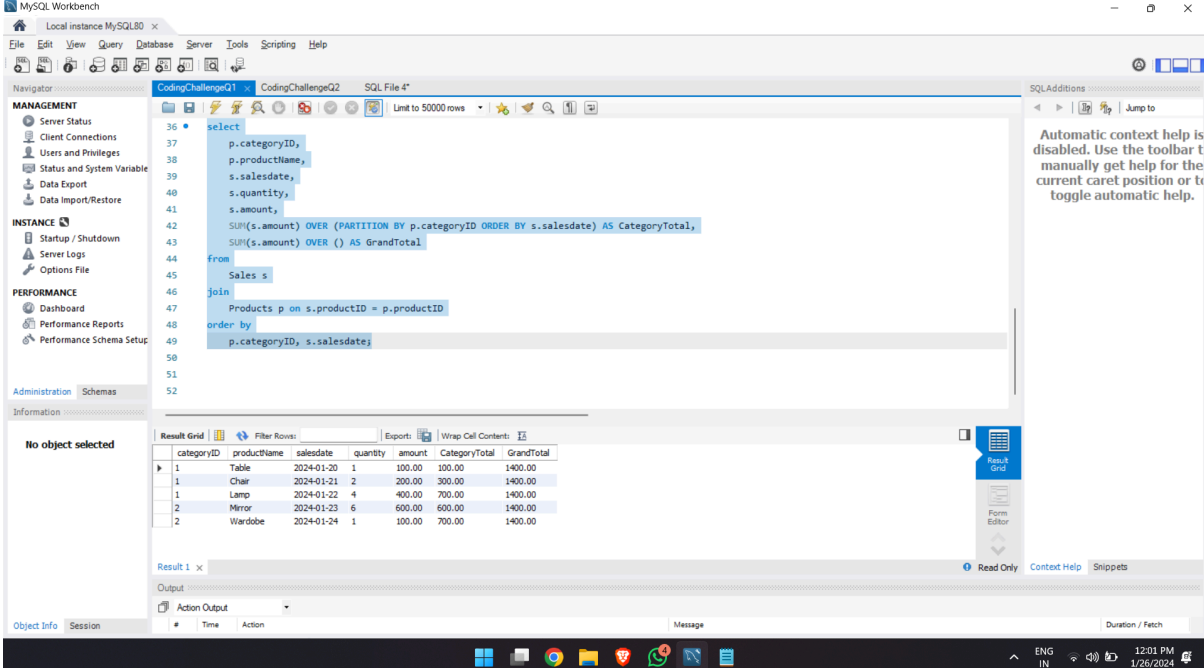


- 1.FROM: Specifies the tables involved in the query (Locations and Payments).
- 2.JOIN: Combines rows from both tables based on the common LocationID.
- 3.GROUP BY: Groups the results by LocationID and LocationName.
- 4.SUM(P.Amount) AS TotalAmount: Calculates the total payment amount for each location.
- 5.HAVING: Filters the grouped results, only including locations where the total amount is greater than 100.
- 6.ORDER BY: Orders the results by the total amount in descending order.
- 7.LIMIT: Limits the number of results to 5.

How to calculate Subtotals in SQL Queries :

aggregate_function(attribute) **OVER** (**PARTITION BY** attribute)

So to get total Sales done CategoryID wise i have executed a query :



The screenshot shows the MySQL Workbench interface. The SQL editor contains the following query:

```
select
  p.categoryID,
  p.productName,
  s.salesdate,
  s.quantity,
  s.amount,
  SUM(s.amount) OVER (PARTITION BY p.categoryID ORDER BY s.salesdate) AS CategoryTotal,
  SUM(s.amount) OVER () AS GrandTotal
from
  Sales s
join
  Products p on s.productID = p.productID
order by
  p.categoryID, s.salesdate;
```

The Results Grid shows the following data:

	categoryID	productName	salesdate	quantity	amount	CategoryTotal	GrandTotal
1	1	Table	2024-01-20	1	100.00	100.00	1400.00
1	1	Chair	2024-01-21	2	200.00	300.00	1400.00
1	1	Lamp	2024-01-22	4	400.00	700.00	1400.00
2	2	Mirror	2024-01-23	6	600.00	600.00	1400.00
2	2	Wardrobe	2024-01-24	1	100.00	700.00	1400.00

SUM(s.amount) OVER (PARTITION BY p.categoryID ORDER BY s.salesdate) AS CategoryTotal

In the above statement **SUM()** is an aggregate function which gives the total of that particular row.

PARTITION BY is used to divide table on attribute categoryID which will find total amount of that particular category.

SUM(s.amount) OVER () AS GrandTotal

And above statement is to calculate the total sales which is referred as GrandTotal.

So we can see

SubTotal of CategoryID 1 = 100+200+400 = 700

SubTotal of CategoryID 2 = 600+100 = 700

Grand Total i.e Category 1 + Category 2 = 700 +700 = 1400

Differences Between UNION EXCEPT and INTERSECT Operators in SQL Server :

The UNION, EXCEPT, and INTERSECT operators are used for combining or comparing the results of two or more SELECT queries in SQL Server. Here are the key differences between them:

1. UNION Operator:

Purpose:

UNION is used to combine the results of two or more SELECT statements into a single result set.

It removes duplicate rows from the result set.

Syntax:

```
SELECT column1, column2, ...  
FROM table1  
UNION  
SELECT column1, column2, ...  
FROM table2;
```

Behaviour:

It combines rows from both queries into a single result set.

It removes duplicate rows.

2. EXCEPT Operator:

Purpose:

EXCEPT is used to return distinct rows from the first SELECT statement that are not present in the result of the second SELECT statement.

It effectively subtracts the result set of the second query from the result set of the first query.

Syntax:

```
SELECT column1, column2, ...  
FROM table1  
EXCEPT  
SELECT column1, column2, ...  
FROM table2;
```


Behaviour:

It returns rows that are unique to the first query and not present in the second query.

It removes duplicate rows from the result.

3. INTERSECT Operator:

Purpose:

INTERSECT is used to return distinct rows that are common to both SELECT statements.

It returns the intersection of the result sets.

Syntax:

```
SELECT column1, column2, ...
```

```
FROM table1
```

```
INTERSECT
```

```
SELECT column1, column2, ...
```

```
FROM table2;
```

Behaviour:

It returns rows that are common to both queries.

It removes duplicate rows from the result.

STAR SCHEMA :

Definition:

The Star Schema is a denormalized schema design where data is organised into a central fact table surrounded by dimension tables. It simplifies the structure by minimising the number of tables and avoiding excessive normalisation.

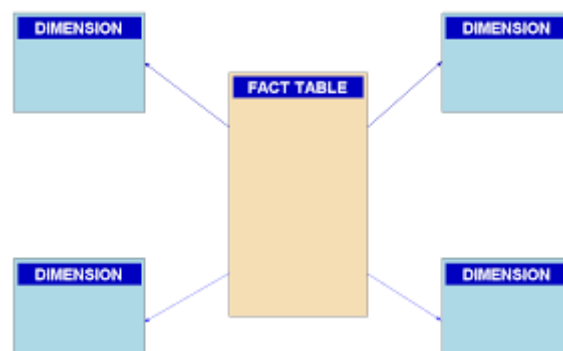
Characteristics:

Consists of a central fact table and dimension tables connected to it.

Each dimension is represented by a single table.

Fact table contains numerical performance metrics (e.g., sales, revenue) and foreign keys referencing dimension tables.

Relationships between fact and dimension tables are straightforward.



SNOWFLAKE SCHEMA :

Definition:

The Snowflake Schema is a normalised form of a Star Schema. It structures data in a way that eliminates redundancy by breaking down dimension tables into multiple related tables.

Characteristics:

Dimensions are organised into multiple related normalised tables. Each level of hierarchy in a dimension is represented by a separate table. It allows for more efficient use of storage space due to normalisation. Relationships between tables are maintained through foreign key relationships.

