**Interview Questions**

# Problem 1:

# Find the distance travelled by each car per day.

Suppose you have a car travelling certain distance and the data is presented as follows:

Day 1 - 50 Kms

Day 2 - 100 Kms

Day 3 - 200 Kms

Now the distance is a cumulative sum as in:

Row 2 = (Kms travelled by car on that day + Row 1 kms)

How should we create a table in the form of Kms travelled by the car on a given day and not the sum of the total distance?

**select \*,**

**(cumulative\_distance - lag(cumulative\_distance, 1, 0)**

**over (partition by cars order by cars))**

**as distance\_travelled**

**from car\_travels;**

## Query Explanation:

The query calculates the distance traveled by each car on a specific day using cumulative distance data from the car\_travels table.

## Query Breakdown:

**select \*: Selects all columns from the car\_travels table.**

(cumulative\_distance - lag(cumulative\_distance, 1, 0) over (partition by cars order by cars)):

**lag(cumulative\_distance, 1, 0):**

Retrieves the cumulative distance of the previous row for the same car (partition by cars).

If no previous row exists (e.g., Day 1), it defaults to 0 (third parameter).

**cumulative\_distance - lag(...):**

Subtracts the previous day's cumulative distance from the current day's cumulative distance to calculate the distance travelled on that specific day.

**as distance\_travelled:**

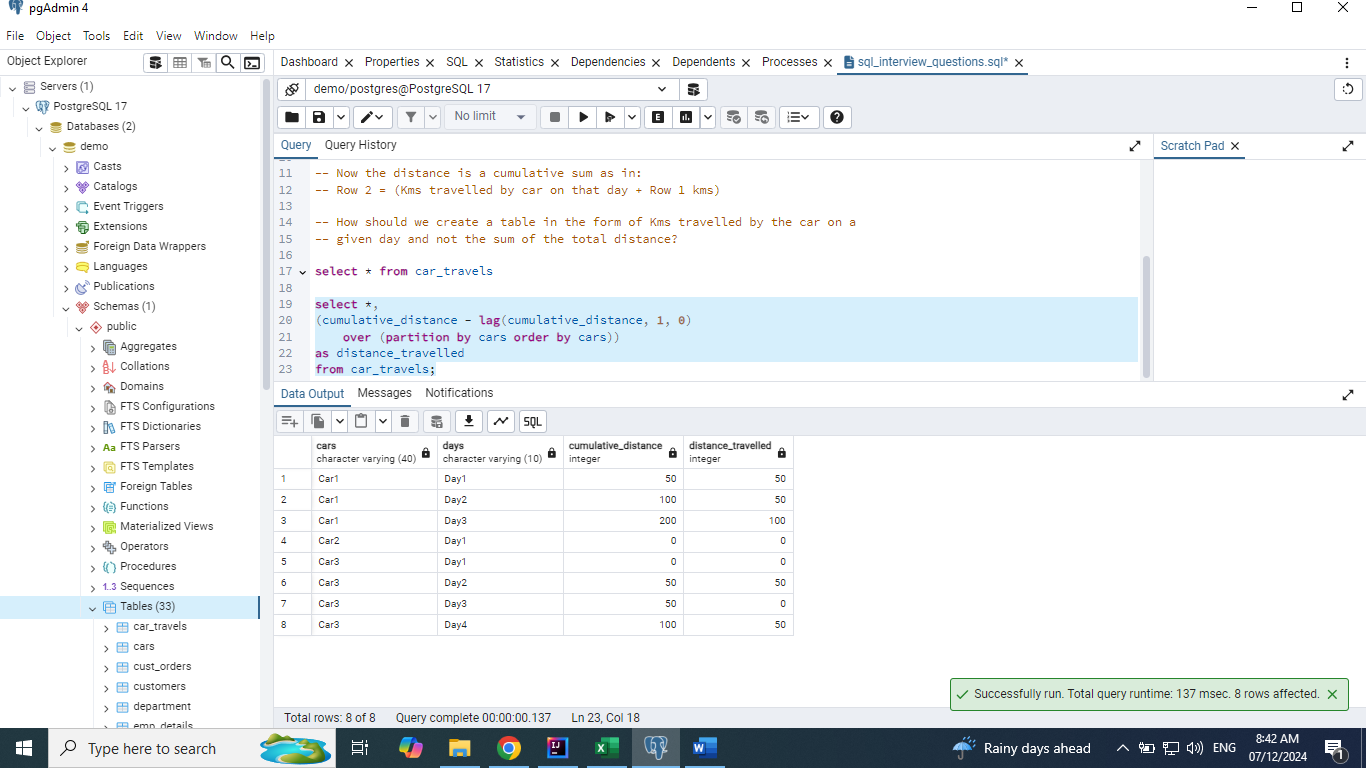
Renames the result of the calculation to distance\_travelled.

**Window Function:**

**over (partition by cars order by cars):**

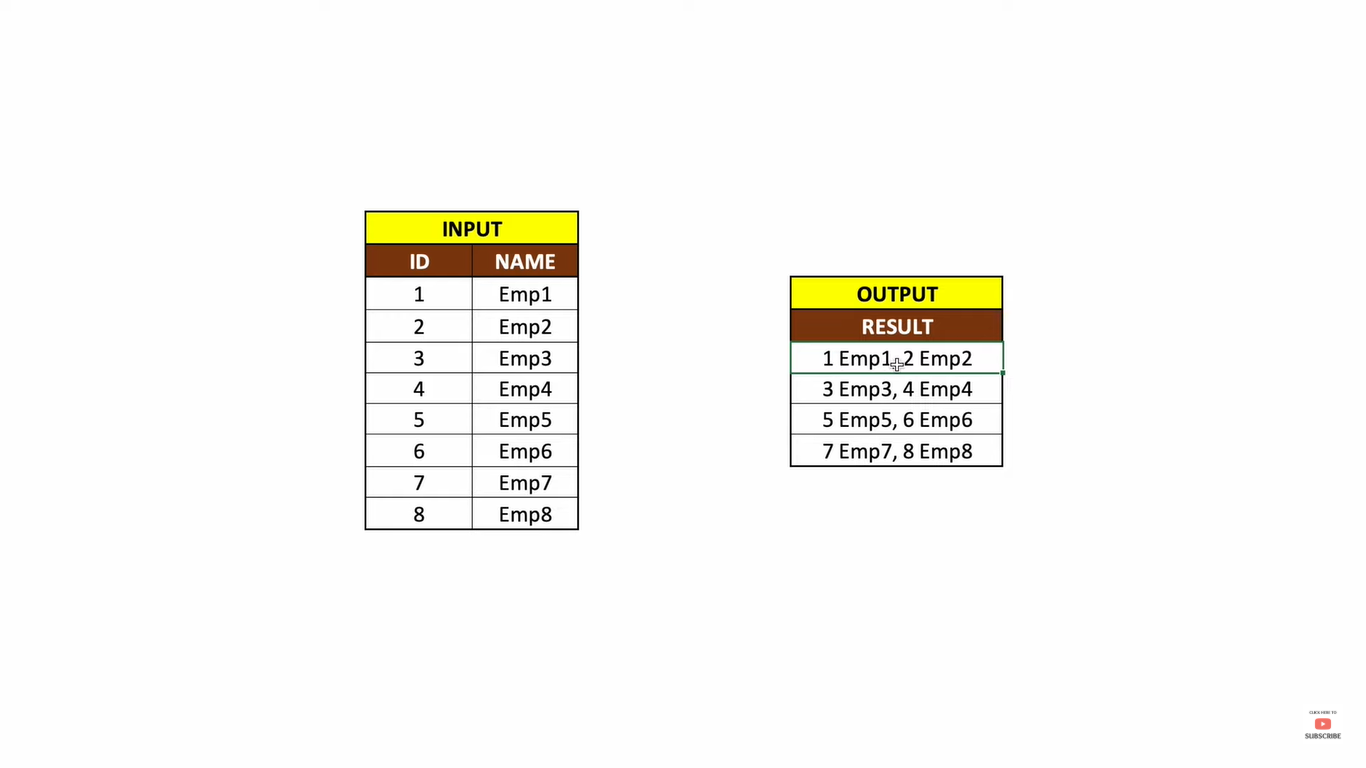
Divides the data by each car (partition by cars), ensuring calculations are performed within the scope of a single car.

Orders the rows for each car to process the cumulative distances in sequence.



# Problem 2:

# Convert the Row level data to Column level data. Also, Combine the data of pair of rows using comma.



**select \*,**

**ntile(4) over(order by id) as buckets**

**from emp\_input;**

**select concat(id,' ', name) as name,**

**ntile(4) over(order by id) as buckets**

**from emp\_input;**

**with cte as**

**(select concat(id,' ', name) as name,**

**ntile(4) over(order by id) as buckets**

**from emp\_input)**

**select string\_agg(name, ', ') as final\_result**

**from cte**

**group by buckets**

**order by 1;**

## Query Explanation:

The query divides employees into 4 equal-sized groups (buckets) based on their id and then concatenates their id and name into a comma-separated string for each bucket.

## Query Breakdown:

**Step 1: Common Table Expression (CTE)**

with cte as (

select

concat(id, ' ', name) as name,

ntile(4) over (order by id) as buckets

from emp\_input

)

**concat(id, ' ', name) as name:**

Combines the id and name of each employee into a single string, formatted as "id name".

For example: 1 Emp1, 2 Emp2, etc.

**ntile(4) over (order by id):**

Divides the employees into 4 groups (buckets).

The ntile function evenly distributes rows across the specified number of groups.

Groups are formed based on the id in ascending order.

**Step 2: Aggregating and Grouping**

select

string\_agg(name, ', ') as final\_result

from cte

group by buckets

order by 1;

string\_agg(name, ', '):

Aggregates all name values in each bucket into a single comma-separated string.

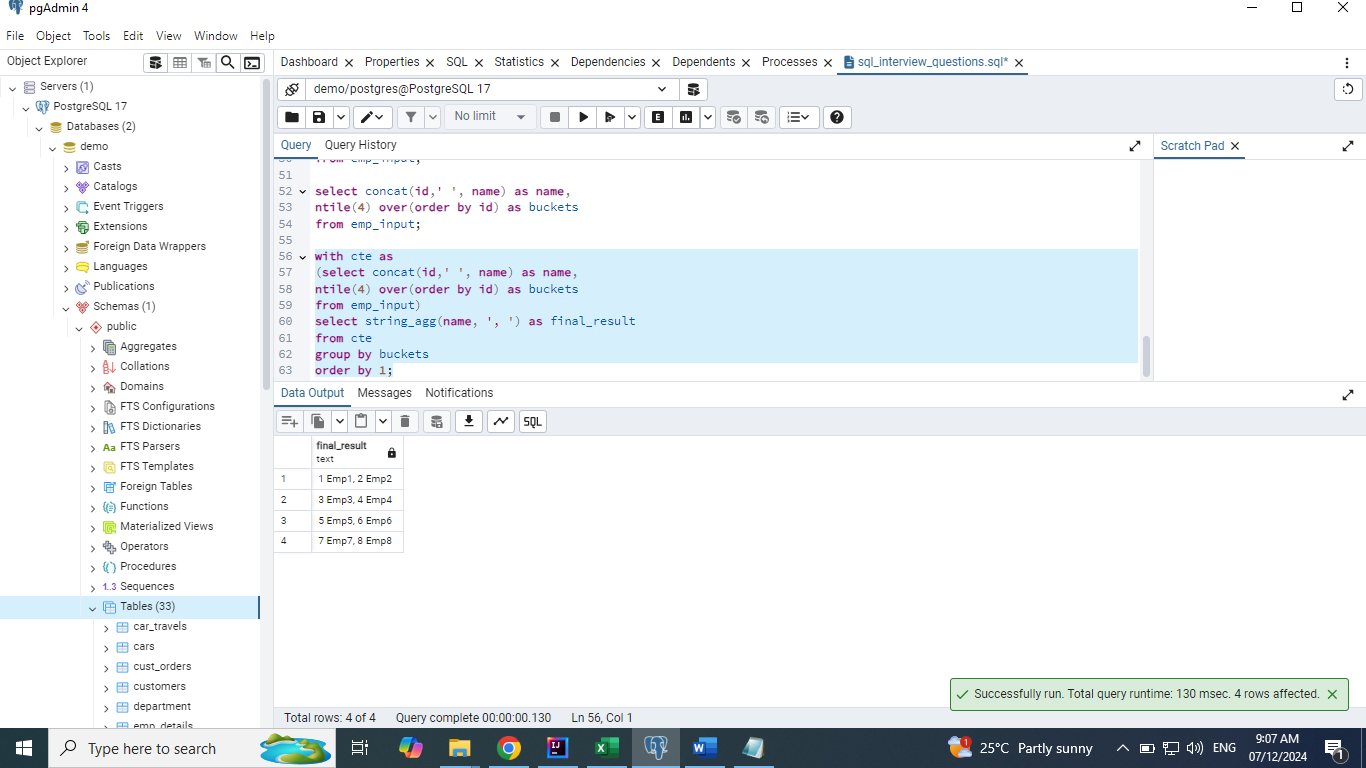
For example: "1 Emp1, 2 Emp2" for the first bucket.

group by buckets:

Groups rows by their assigned bucket, ensuring that string\_agg works within each bucket.

order by 1:

Orders the final results by the aggregated string.



# Problem 3:

# Write a solution to report the type of each node in the tree. Return the result table in any order.

Table: Tree

+-------------+------+

| Column Name | Type |

+-------------+------+

| id | int |

| p\_id | int |

+-------------+------+

id is the column with unique values for this table.

Each row of this table contains information about the id of a node and the id of its parent node in a tree.

The given structure is always a valid tree.

Each node in the tree can be one of three types:

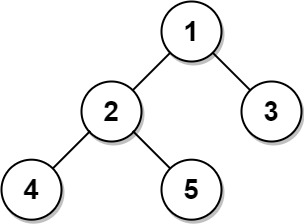
* **"Leaf"**: if the node is a leaf node.
* **"Root"**: if the node is the root of the tree.
* **"Inner"**: If the node is neither a leaf node nor a root node.

Write a solution to report the type of each node in the tree.

Return the result table in **any order**.

The result format is in the following example.

**Example 1:**



**Input:**

Tree table:

+----+------+

| id | p\_id |

+----+------+

| 1 | null |

| 2 | 1 |

| 3 | 1 |

| 4 | 2 |

| 5 | 2 |

+----+------+

**Output:**

+----+-------+

| id | type |

+----+-------+

| 1 | Root |

| 2 | Inner |

| 3 | Leaf |

| 4 | Leaf |

| 5 | Leaf |

+----+-------+

**Explanation:**

Node 1 is the root node because its parent node is null and it has child nodes 2 and 3.

Node 2 is an inner node because it has parent node 1 and child node 4 and 5.

Nodes 3, 4, and 5 are leaf nodes because they have parent nodes and they do not have child nodes.

**Example 2:**



**Input:**

Tree table:

+----+------+

| id | p\_id |

+----+------+

| 1 | null |

+----+------+

**Output:**

+----+-------+

| id | type |

+----+-------+

| 1 | Root |

+----+-------+

**Explanation:** If there is only one node on the tree, you only need to output its root attributes.

## Query Explanation:

This query determines the type of each node in a tree structure based on its relationships with parent and child nodes. The type of each node is classified as Root, Inner, or Leaf.

## Query Breakdown:

**Basic Selection:**

select \*

from tree;

Selects all columns (id, p\_id) from the tree table.

**Type Classification with CASE:**

case

when p\_id is null then 'Root'

when p\_id is not null and id in (select distinct p\_id from tree) then 'Inner'

else 'Leaf'

end as type

**Condition 1:**

**when p\_id is null then 'Root'**

A node is classified as Root if its p\_id (parent ID) is null. This indicates the node has no parent and is the top-most node in the tree.

**Condition 2:**

**when p\_id is not null and id in (select distinct p\_id from tree) then 'Inner'**

A node is classified as Inner if:

It has a parent (p\_id is not null).

It is a parent of other nodes (id in (select distinct p\_id from tree)). This checks whether the id appears in the p\_id column of any row in the tree table.

**Condition 3:**

else 'Leaf'

A node is classified as Leaf if:

It has a parent (p\_id is not null).

It is not a parent of any other node. In this case, the id does not appear in the p\_id column of the tree table.

**select \*,**

**case**

**when p\_id is null then 'Root'**

**when p\_id is not null and id in (select distinct p\_id from tree) then 'Inner'**

**else 'Leaf'**

**end as type**

**from tree;**

