**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:**

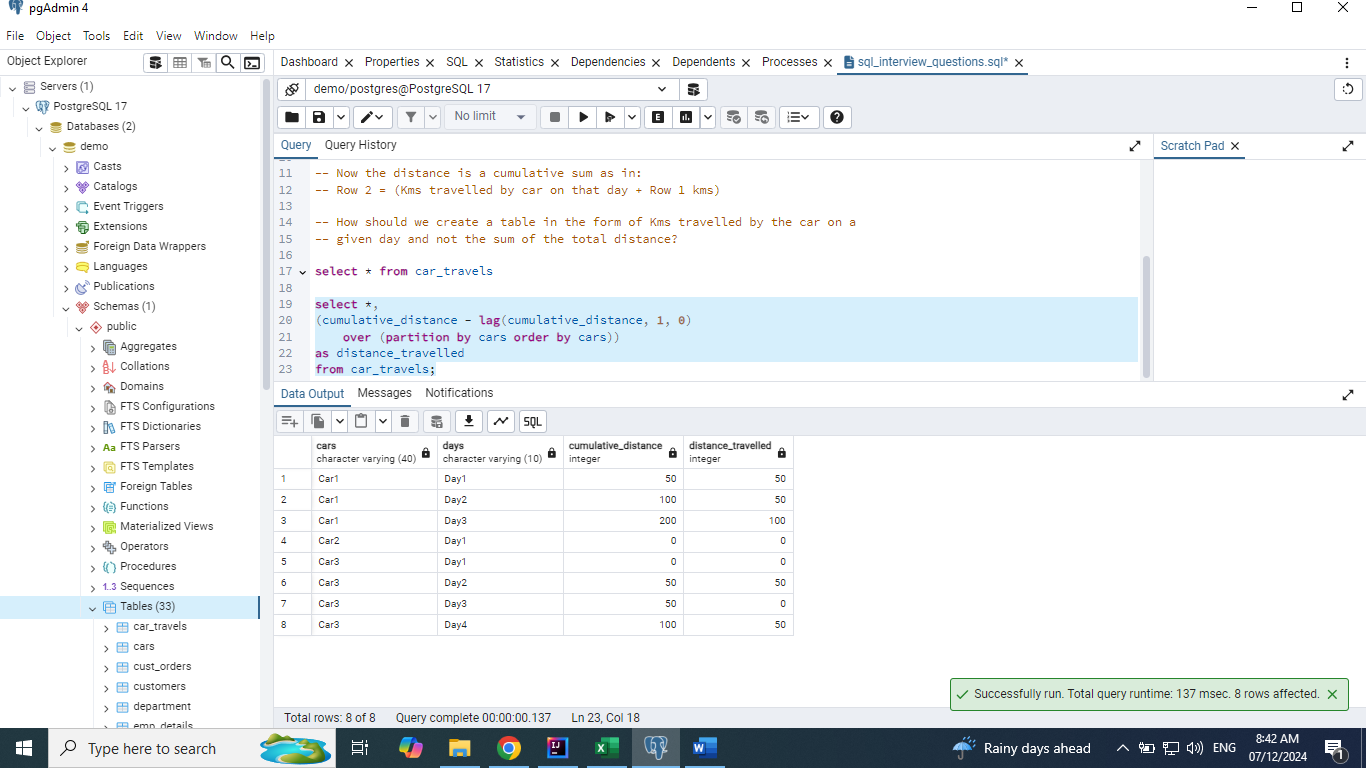
Renames the result of the calculation to distance\_travelled.

**Window Function:**

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

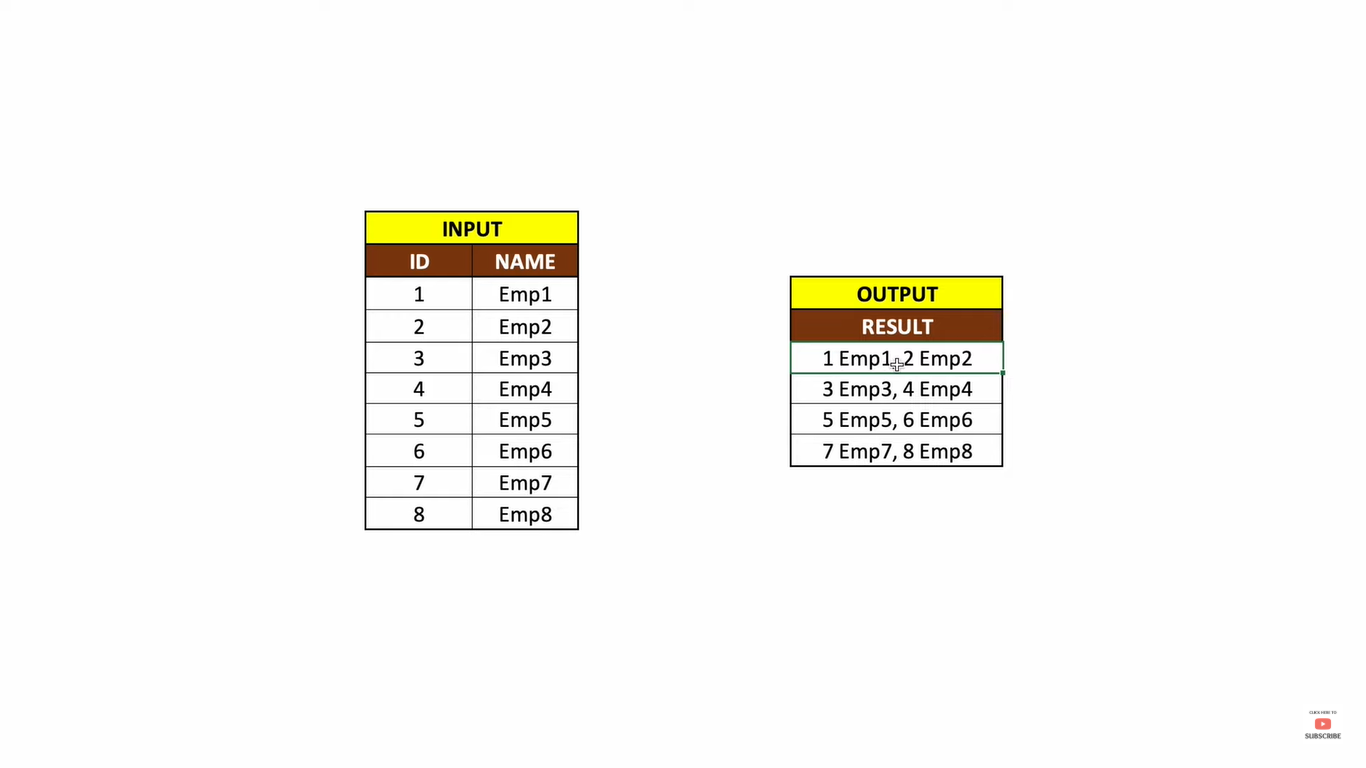
Divides the data by each car (partition by cars), ensuring calculations are performed within the scope of a single car.

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.

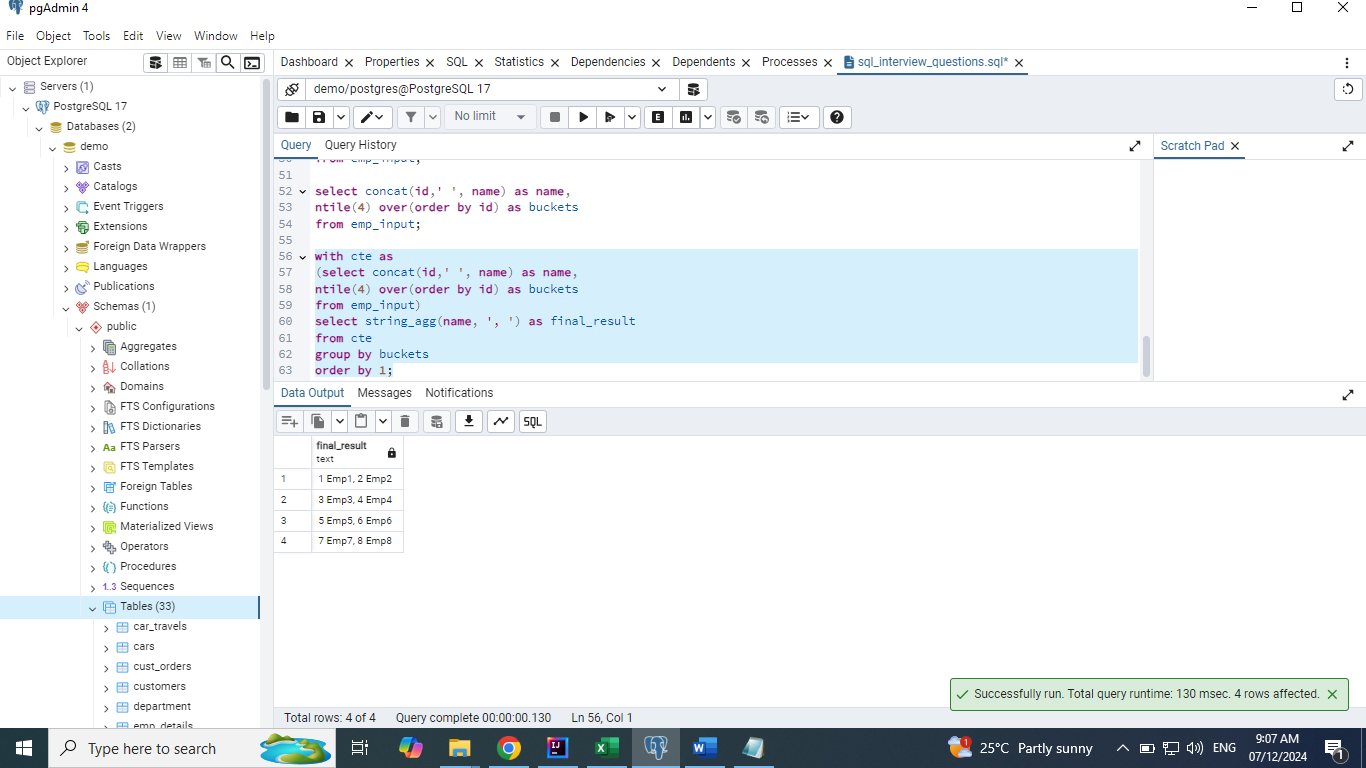
For example: "1 Emp1, 2 Emp2" for the first bucket.

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:

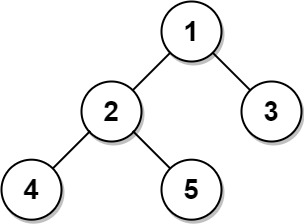
* **"Leaf"**: if the node is a leaf node.
* **"Root"**: if the node is the root of the tree.
* **"Inner"**: If the node is neither a leaf node nor a root node.

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**

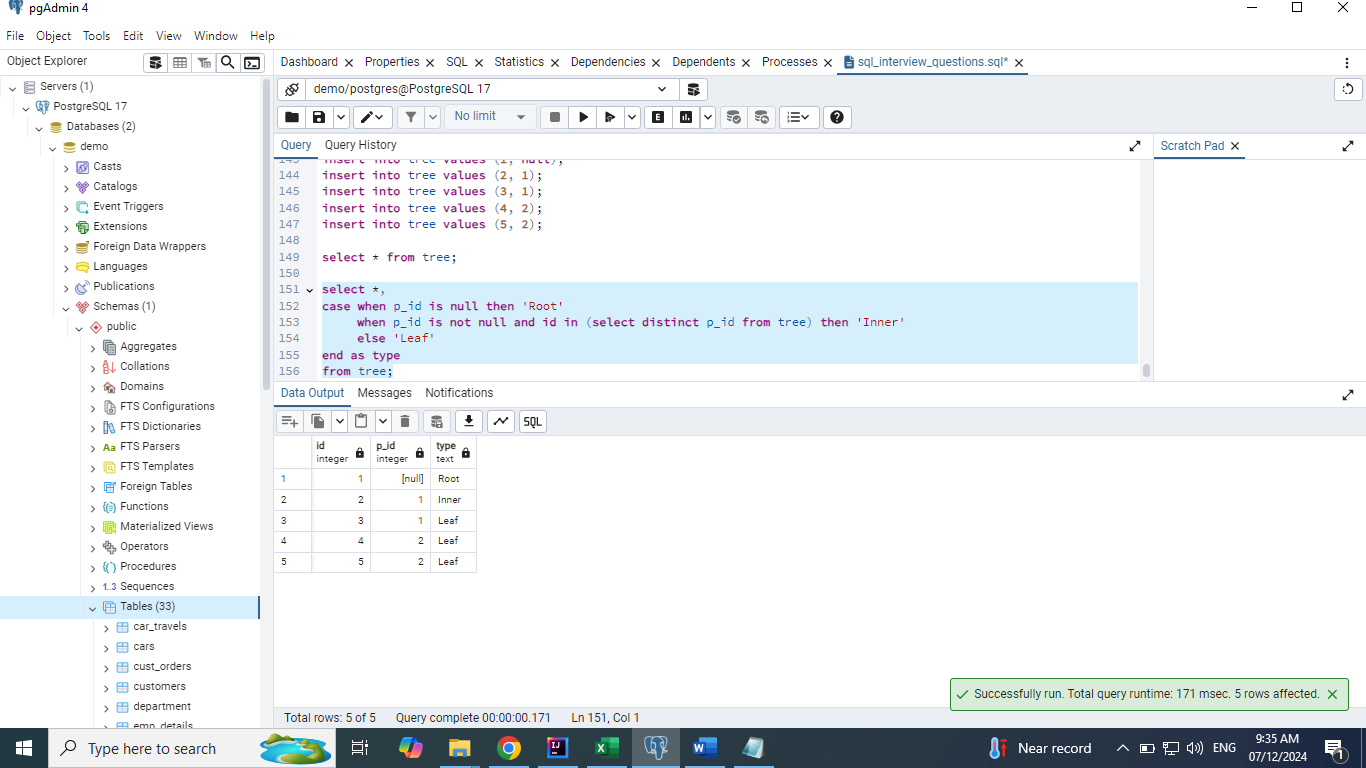
**when p\_id is null then 'Root'**

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

**else 'Leaf'**

**end as type**

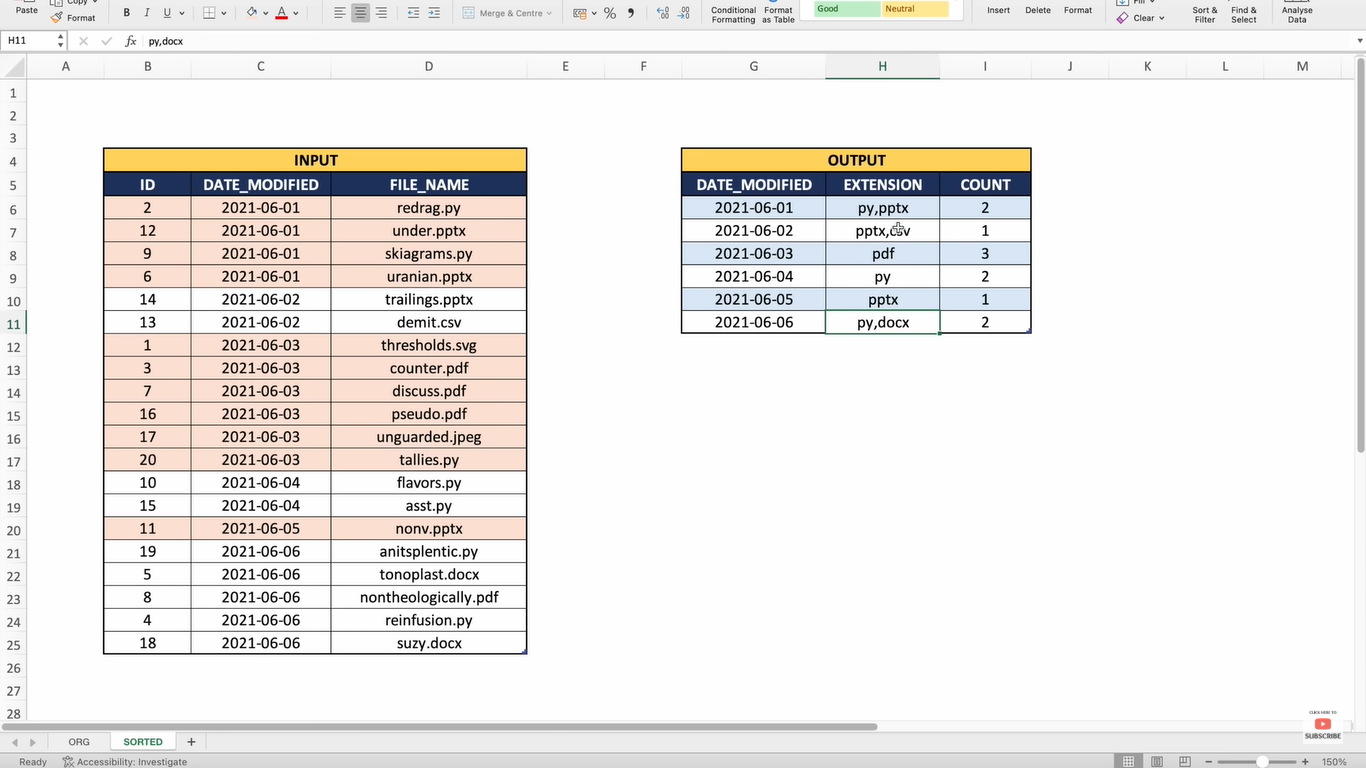
**from tree;**



# Problem 4:

# Find the most modified file extension for the day

A DB contains a list of filenames including their extensions and the dates they were last modified. For each date that a modification was made, return the date, the extension(s) of the files that were modified the most, and the number of files modified that date. If more than one file extension ties for the most modifications, return them as a comma delimited list in reverse order.



Problem Breakdown

Fetch the file extension

select \* , position('.' in file\_name)

from files;

select date\_modified, substring(file\_name, (position('.' in file\_name)+1)) as file\_ext

from files;

For each day, how many times each file extension was modified

select date\_modified, substring(file\_name, (position('.' in file\_name)+1)) as file\_ext, count(1) as cnt

from files

group by date\_modified, file\_ext

order by 1;

For each day, fetch the most modified file extension from #2

with cte as

(select date\_modified, substring(file\_name, (position('.' in file\_name)+1)) as file\_ext, count(1) as cnt

from files

group by date\_modified, file\_ext

order by 1)

select \*

from cte c1

where cnt = (select max(cnt) from cte c2

where c2.date\_modified = c1.date\_modified);

If there is tie, then concatenate the multiple file extension

## Query Explanation:

The given query identifies the file extensions that were most frequently modified on each date from a database of files. It returns the following for each modification date:

1. The **modification date**.
2. The **file extension(s)** with the highest count of modifications for that date (comma-separated in reverse alphabetical order if there's a tie).
3. The **number of files modified** for the most frequent file extension(s) on that date.

## Query Breakdown:

**Extract File Extension and Count by Date:**

A Common Table Expression (cte) is created.

For each date\_modified, the query extracts the file extension using substring(file\_name, (position('.' in file\_name) + 1)).

It counts the number of files modified for each date\_modified and file\_ext combination.

The result is grouped by date\_modified and file\_ext, and ordered by the modification date

**Find Extensions with Maximum Modifications Per Date:**

The main query iterates through each row of the cte.

It compares the cnt of each file extension with the maximum cnt for that date\_modified using a subquery.

If the cnt equals the maximum, the extension is considered one of the most modified for that date.

**Aggregate Extensions in Reverse Alphabetical Order:**

For each date\_modified, extensions meeting the maximum count criteria are concatenated into a single string using string\_agg with ORDER BY file\_ext DESC.

**Final Output:**

The query groups the results by date\_modified.

**It selects:**

date\_modified.

A comma-separated list of extensions in reverse alphabetical order (extension).

The maximum count of modified files for that date (count).

**with cte as**

**(select date\_modified, substring(file\_name, (position('.' in file\_name)+1)) as file\_ext, count(1) as cnt**

**from files**

**group by date\_modified, file\_ext**

**order by 1)**

**select date\_modified,**

**string\_agg(file\_ext, ', ' order by file\_ext desc) as extension,**

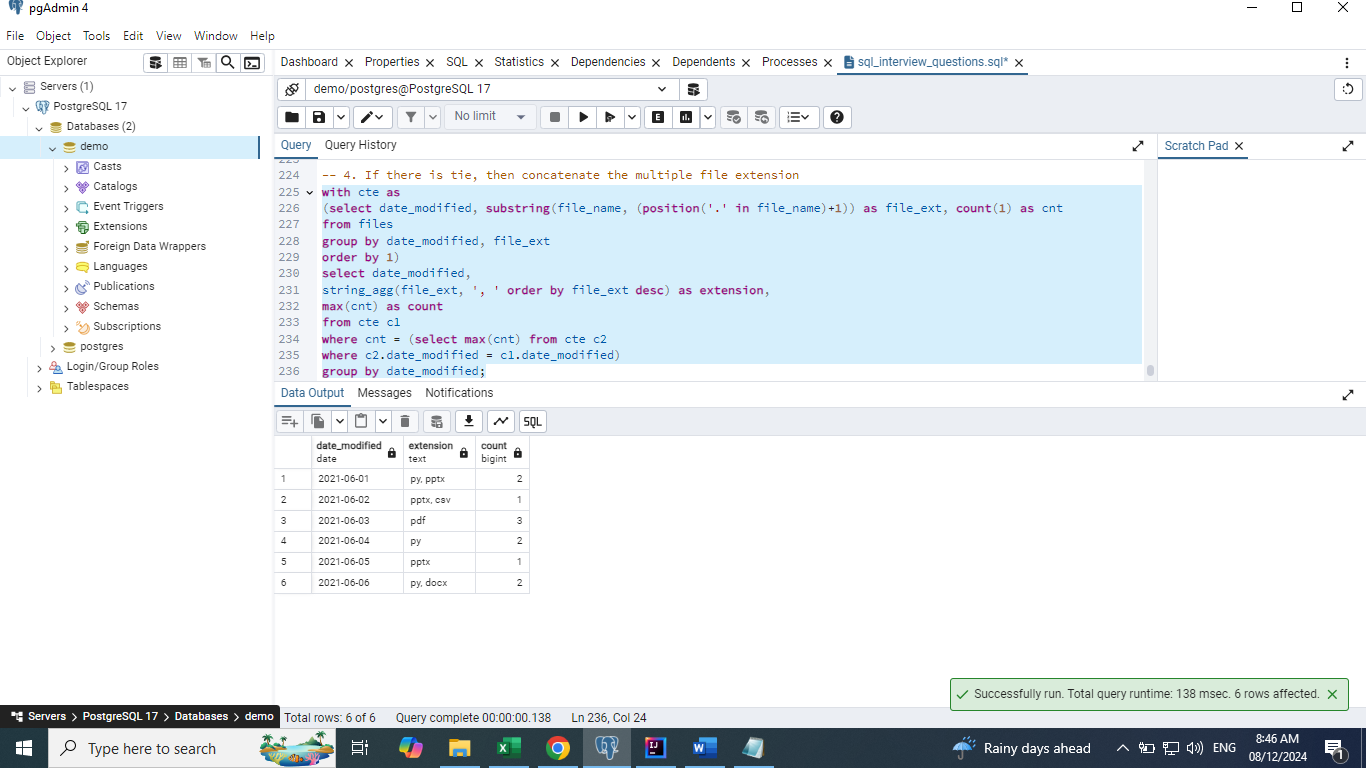
**max(cnt) as count**

**from cte c1**

**where cnt = (select max(cnt) from cte c2**

**where c2.date\_modified = c1.date\_modified)**

**group by date\_modified;**



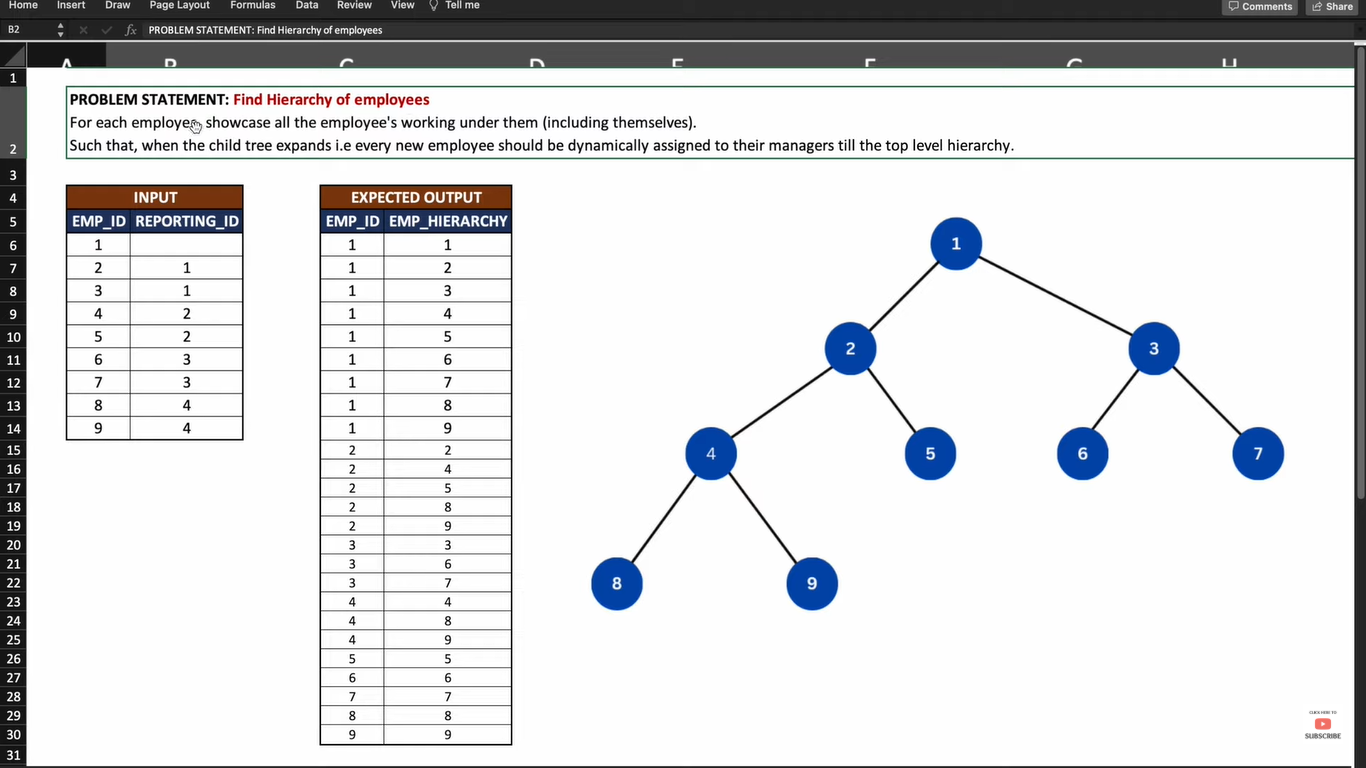
# Problem 5:

Find the hierarchy of employees

For each employee, showcase all the employee's working under them (including themselves)

Such that, when the child tree expands i.e., every new employee should be dynamically

assigned to their managers till the top-level hierarchy.



## Recursive SQL syntax

**with recursive cte as**

**(base query**

**union all**

**recursive part of the query**

**termination / exit condition)**

**select \***

**from cte;**

## Query Explanation:

The query generates the **hierarchy of employees** from a table, emp\_hierarchy, where each row defines an employee and their manager (or reporting\_id). For each employee, it lists:

* The employee themselves.
* All employees working under them, either directly or indirectly, forming a tree-like structure.

## Query Breakdown:

**1. Recursive Common Table Expression (CTE):**

The query uses a WITH RECURSIVE clause to define a CTE, called cte, to compute the hierarchy dynamically.

with recursive cte as

(

select emp\_id, emp\_id as employee\_hierarchy

from emp\_hierarchy

union all

select cte.emp\_id, eh.emp\_id as employee\_hierarchy

from cte

join emp\_hierarchy eh on cte.employee\_hierarchy = eh.reporting\_id

)

**Base Case:**

select emp\_id, emp\_id as employee\_hierarchy

from emp\_hierarchy

Starts with the base row for each emp\_id.

Here, employee\_hierarchy is initialized to the emp\_id itself, meaning each employee is their own manager.

**Recursive Step:**

select cte.emp\_id, eh.emp\_id as employee\_hierarchy

from cte

join emp\_hierarchy eh on cte.employee\_hierarchy = eh.reporting\_id

The recursive step finds all employees who report to someone already in the hierarchy.

It iterates through the hierarchy to add employees working under the emp\_id in the CTE.

**2. Final Output:**

The select statement retrieves the hierarchy:

select \*

from cte

order by emp\_id, employee\_hierarchy;

For each emp\_id, it lists their entire hierarchy, including themselves, sorted by emp\_id and employee\_hierarchy.

**with recursive cte as**

**(select emp\_id, emp\_id as employee\_hierarchy**

**from emp\_hierarchy**

**union all**

**select cte.emp\_id, eh.emp\_id as employee\_hierarchy**

**from cte**

**join emp\_hierarchy eh on cte.employee\_hierarchy = eh.reporting\_id)**

**select \***

**from cte**

**order by emp\_id, employee\_hierarchy;**

**1st Iteration**

**select emp\_id, emp\_id as employee\_hierarchy**

**from emp\_hierarchy where emp\_id = 1;**

**2nd Iteration**

**select cte.emp\_id, eh.emp\_id as employee\_hierarchy**

**from (select emp\_id, emp\_id as employee\_hierarchy**

**from emp\_hierarchy where emp\_id = 1) cte**

**join emp\_hierarchy eh on cte.employee\_hierarchy = eh.reporting\_id;**

**3rd Iteration**

**select cte.emp\_id, eh.emp\_id as employee\_hierarchy**

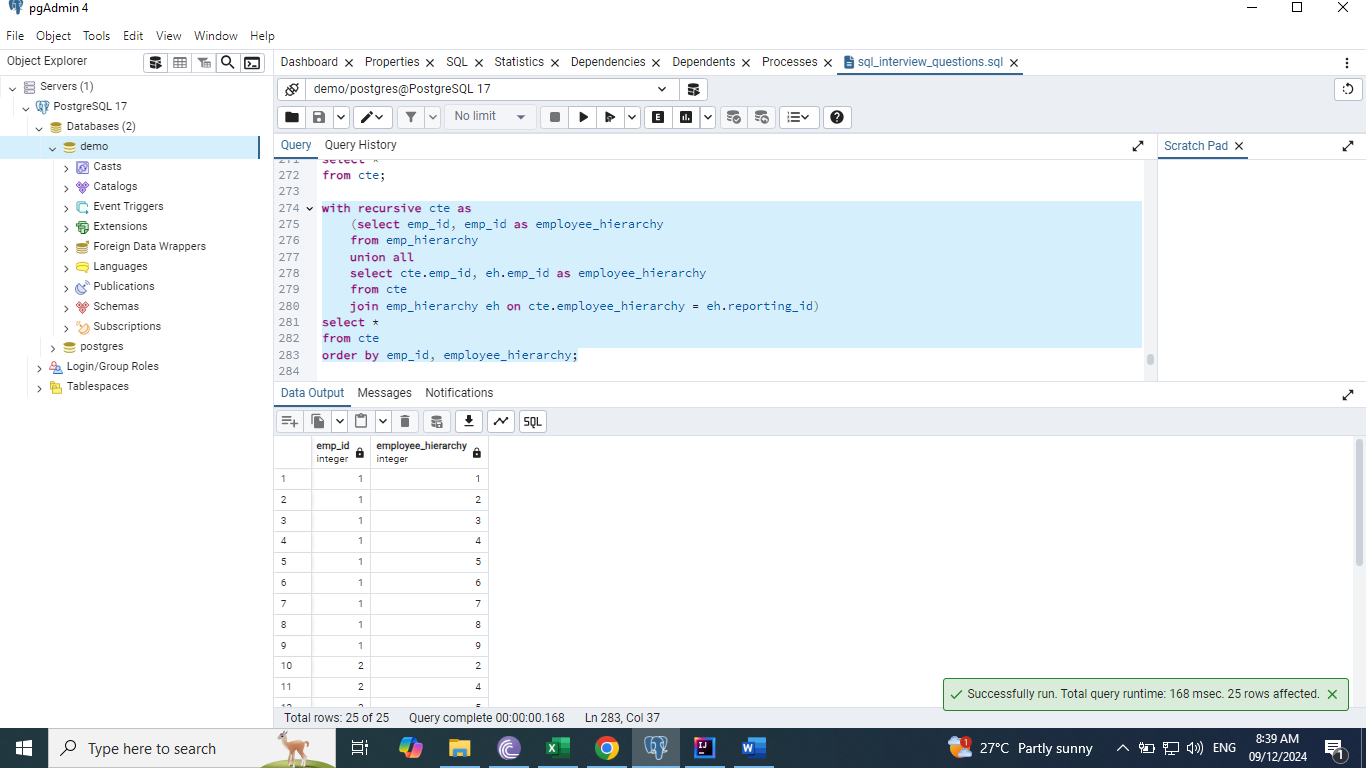
**from (select cte.emp\_id, eh.emp\_id as employee\_hierarchy**

**from (select emp\_id, emp\_id as employee\_hierarchy**

**from emp\_hierarchy where emp\_id = 1) cte**

**join emp\_hierarchy eh on cte.emp\_id = eh.reporting\_id)cte**

**join emp\_hierarchy eh on cte.employee\_hierarchy = eh.reporting\_id;**



# Problem 6:

Given is a list of arbitrary values. They can either be comma separated values in a single row column or they could be values spread across multiple rows. Write an SQL Query to group these arbitrary values as per the expected output.

**Step by Step approach**

**Convert the column level data to row level data**

select unnest(val) from arbitrary\_values;

**Adding unique identifier with ordinality**

select x.\*

from arbitrary\_values

cross join unnest(val) with ordinality x(val, index);

## Query Explanation:

This SQL query transforms a single-column array of values into grouped rows where each group size grows incrementally. Here's a step-by-step explanation:

## Query Breakdown:

**Step 1: Explode the Array Column**

The array val is expanded into rows using the unnest function:

**SELECT unnest(val) FROM arbitrary\_values;**

This converts an array like "{a1,a2,a3,...}" into a single column of values (a1, a2, etc.) in individual rows.

**Step 2: Add Ordinality**

To maintain the original order and allow indexing, unnest is combined with the WITH ORDINALITY option:

**SELECT x.\***

**FROM arbitrary\_values**

**CROSS JOIN UNNEST(val) WITH ORDINALITY x(val, index);**

val: The individual values from the array.

index: The position of the value in the original array.

This gives rows like:

val index

a1 1

a2 2

a3 3

... ...

This result is stored in the cte\_values Common Table Expression (CTE).

**Step 3: Create Recursive Groups**

Using a Recursive CTE, the query builds groups with sizes that increase incrementally. Here's the breakdown:

**3.1. Base Case**

The first row (index = 1) initializes the recursion:

**SELECT \*, 1 AS iterator, MAX(index) OVER() AS max\_index**

**FROM cte\_values**

**WHERE index = 1;**

iterator = 1: Group number starts at 1.

max\_index: Stores the highest index value to track group boundaries.

**3.2. Recursive Case**

Each recursive step processes the next set of rows by expanding the group size by one:

**SELECT cv.\*, (iterator + 1), MAX(cv.index) OVER() AS max\_index**

**FROM cte**

**JOIN cte\_values cv**

**ON cv.index BETWEEN max\_index + 1 AND max\_index + 1 + iterator;**

**cv: Refers to rows in cte\_values.**

iterator + 1: Advances to the next group.

index BETWEEN max\_index + 1 AND max\_index + 1 + iterator: Picks the next set of rows for the current group.

**Step 4: Aggregate Values into Groups**

Once grouped, the query aggregates the values in each group into a comma-separated string using string\_agg:

**SELECT iterator AS grp, STRING\_AGG(val, ' , ') AS values**

**FROM cte**

**GROUP BY iterator**

**ORDER BY iterator;**

grp: The group number (from iterator).

values: The aggregated string of values in each group.

**Final Query**

**with recursive cte as**

**(select \*, 1 as iterator, max(index) over() as max\_index**

**from cte\_values where index = 1**

**union**

**select cv.\*, (iterator+1), max(cv.index) over() as max\_index**

**from cte**

**join cte\_values cv on cv.index between max\_index+1 and max\_index+1+iterator**

**),**

**cte\_values as**

**(select x.\***

**from arbitrary\_values**

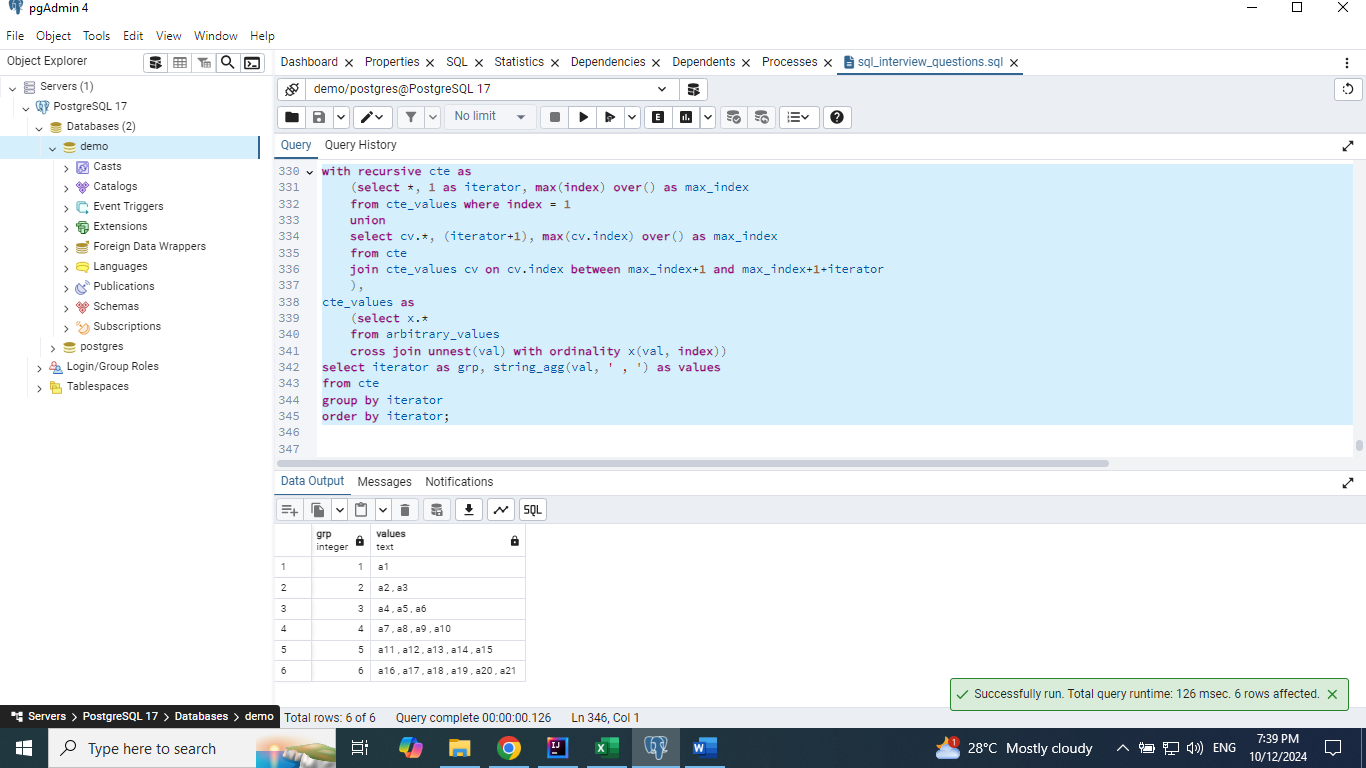
**cross join unnest(val) with ordinality x(val, index))**

**select iterator as grp, string\_agg(val, ' , ') as values**

**from cte**

**group by iterator**

**order by iterator;**



# Problem 7:

## Type1: SQL Query to fetch "N" consecutive records when temperature is below 0.

**Fetch the temperature below 0.**

select \* from weather where temperature < 0;

**Add a unique row number to the data**

select \*, row\_number() over(order by id) as rn

from weather

where temperature < 0;

**Logic to group the data**

select \*, row\_number() over(order by id) as rn,

(id - row\_number() over(order by id)) as difference

from weather

where temperature < 0;

## Query Explanation:

The query identifies consecutive groups of records from the weather table where the temperature is less than 0 and extracts groups of exactly 3 records.

## Query Breakdown:

**The WITH clause, also known as a Common Table Expression (CTE), is used in SQL to create temporary, named result sets that can be referenced within the main query. It helps make SQL queries more readable and modular, especially when dealing with complex logic or when a subquery is used multiple times.**

**CTE t1:**

Purpose: Add additional columns to analyze the consecutive records.

Columns added:

rn: Row number of the record ordered by id.

difference: A computed value (id - row\_number() over(order by id)) that groups records with consecutive id values. This is because consecutive rows in a sequence will have the same difference.

**CTE t2:**

Purpose: Count the number of records in each group (grouped by difference).

Columns added:

no\_of\_records: Total number of records in the group with the same difference.

This is achieved by using a count(\*) over(partition by difference).

**Final Query:**

Filters the rows to include only those groups where no\_of\_records = 3.

Selects the id, city, temperature, and day columns for these rows.

**Final query using with clause**

**with t1 as**

**(select \*, row\_number() over(order by id) as rn,**

**id - (row\_number() over(order by id)) as difference**

**from weather**

**where temperature < 0),**

**t2 as**

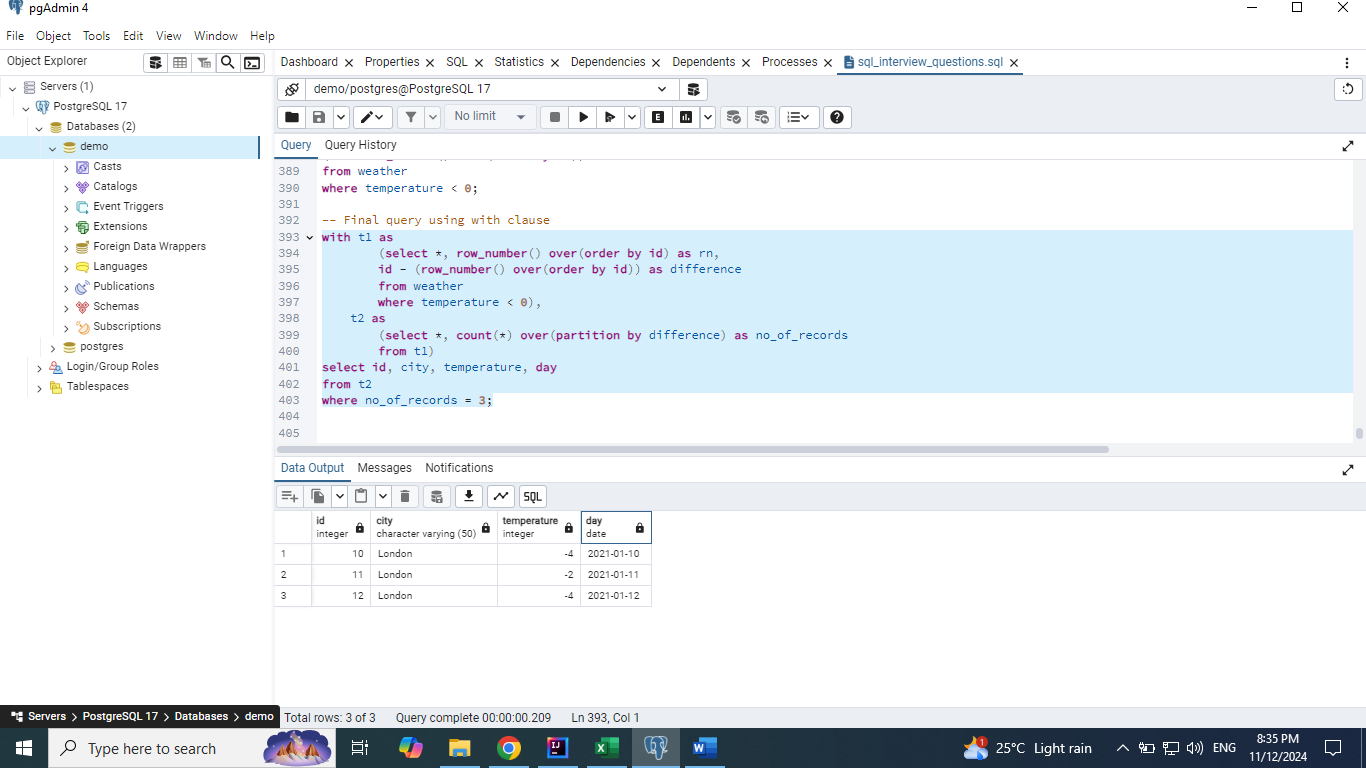
**(select \*, count(\*) over(partition by difference) as no\_of\_records**

**from t1)**

**select id, city, temperature, day**

**from t2**

**where no\_of\_records = 3;**



## Type 2: SQL Query to fetch "N" consecutive records when temperature is below 0 and also table doesn't have id column.

**with w as**

**(select \*, row\_number() over() as id**

**from vw\_weather),**

**t1 as**

**(select \*, row\_number() over(order by id) as rn,**

**id - (row\_number() over(order by id)) as difference**

**from w**

**where temperature < 0),**

**t2 as**

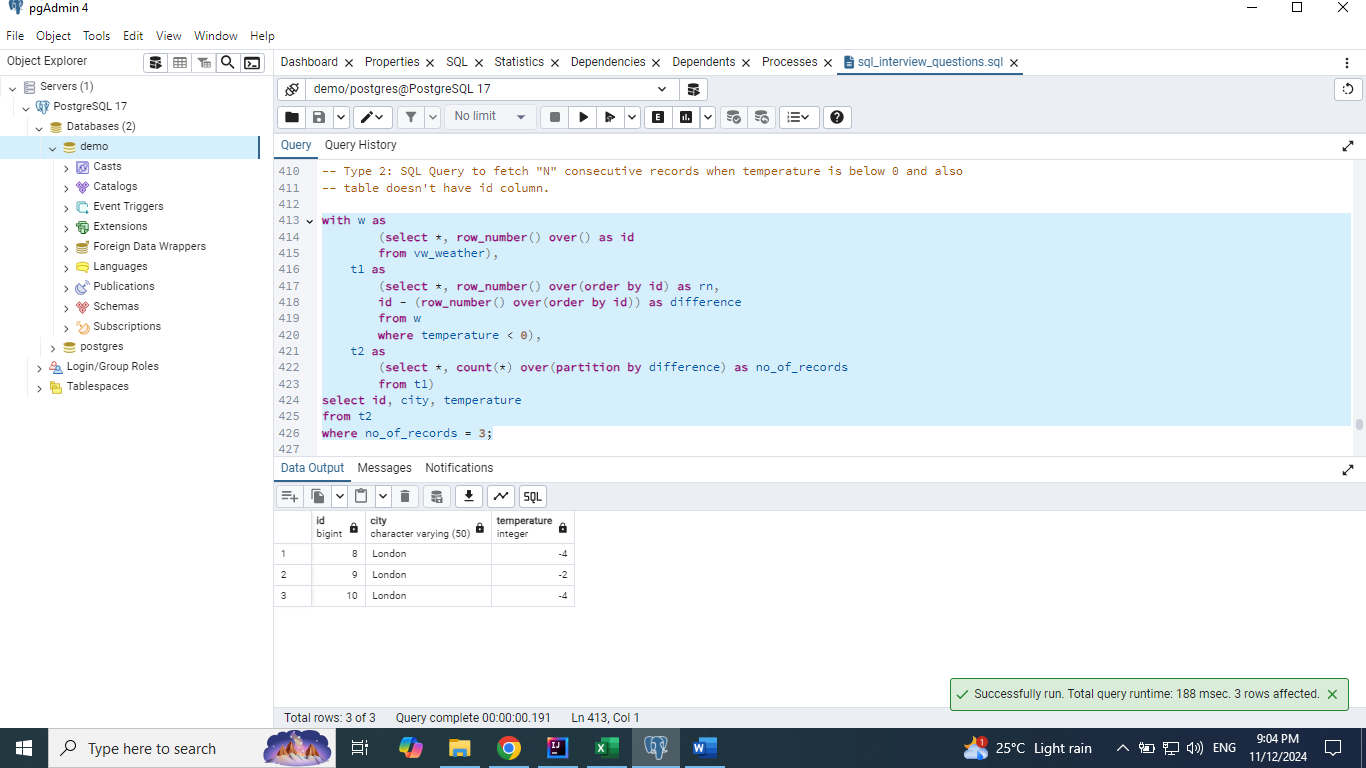
**(select \*, count(\*) over(partition by difference) as no\_of\_records**

**from t1)**

**select id, city, temperature**

**from t2**

**where no\_of\_records = 3;**



## Type 3: Fetch the records from table where there are orders for 3 consecutive days

**select \*, row\_number() over(order by order\_id) as rn**

**from orders;**

**with t1 as**

**(select \*, row\_number() over(order by order\_id) as rn ,**

**order\_date - cast (row\_number() over(order by order\_id)as int) as difference**

**from orders),**

**t2 as**

**(select \*, count(\*) over(partition by difference) as no\_of\_records**

**from t1)**

**select order\_id, order\_date**

**from t2**

**where no\_of\_records = 2;**

