ASSIGNMENT 2 [POSTGRESQL]

WHAT IS THE DIFFERENCE IN BETWEEN VIEW AND MATERIALIZED VIEW.

- VIEW

A view is a virtual table that fetches the data dynamically whenever Queried Which implies Views doesn't store any physical data, Every time we access a view, it runs the underlying query to retrieve the most up-to-date data. It's lead to slower performance due to over Performance head for complex queries.

```
Example use case: We might use a view to simplify complex que
ries or present data in a
more user-friendly way without changing the underlying schem
a.
```

- MATERIALIZED VIEW
  It's stores the results of queries physically, making retrieval more faster but requiring periodic refresh to reflect the changes in underlying data's.

```
Example use case: Materialized views are useful for improving
performance in cases where the underlying data does not chang
e frequently, or for reporting systems where you need to pres
ent a snapshot of data at a specific point in time.
```

Task: Add an index on the DepartmentID column of the Employees table.
Question: Describe how this index could improve query performance.

- Faster Lookups - When we are Quering for Employees belonging to a specific department such as

  `WHERE DepartmentID = :DepartmentID`

  the database can use the INDEX to quickly locate the rows with the matching DepartmentID. Without Index, the database would need to perform a full scan, checking for every row in the table which can be very slow for large tables.
- EFFECTIVE JOIN - If we perform JOIN between Employee and Departments based ON DepartmentID, the INDEX allow PostgreSQL to match the rows very effectively. Instead of scanning all rows, PostgreSQL can use the INDEX to find the matching rows.
- Improve Sorting - If you are ordering the results based on DepartmentID, the INDEX can help the database quickly retrieve data in the correct order without needing to sort the entire tables after fetching the rows.

Explain why refreshing a materialized view is necessary.

Materialized views store the results of a query physically on disk, unlike regular views that retrieve the data dynamically every time they are queried.

If any INSERT, UPDATE, or DELETE operations occur on the base tables that the materialized view depends on, the materialized view will not reflect these changes until it is refreshed. To ensure that the materialized view provides the most up-to-date and accurate information, especially in systems where reports or analysis depend on fresh data.

Since the data in a materialized view is a snapshot of the data at the time it was created or last refreshed, it can become outdated if the underlying tables are modified.

Task: Create a composite index on the Employees table for the columns DepartmentID and Salary.
Question: Explain how this composite index differs from having individual indexes on each column.

A composite index enables the database to optimize queries that involve both DepartmentID and Salary together. This index is particularly useful when queries filter or sort based on both columns, allowing the database to efficiently access rows that meet both conditions. The order of the columns in the composite index matters; in this case, it optimizes queries that filter first by DepartmentID and then by Salary.

On the other hand, individual indexes optimize each column separately. They are beneficial for queries that involve filtering or sorting on only one column at a time. However, when both columns are used in a query, the database may need to perform extra work, such as merging results from multiple indexes, which is less efficient than using a composite index.

Explain how adding an index on the Salary column affects the performance of this query.

When we add an index on the Salary column in the Employees table, it can significantly improve the performance of queries that filter or sort by Salary. Here's how:

1. Consider the following query(With Index) :

```
SELECT *
FROM Employees
WHERE Salary > 50000;
```

Without an index on the Salary column, the database performs a full table scan. This means the database needs to check each row in the table to determine if the Salary is greater than 50,000.

For a large dataset, a full table scan is time-consuming because every row has to be read and evaluated.

2. With an Index on Salary
   Now, after adding the index:

```
CREATE INDEX idx_salary ON Employees(Salary);
```

With an index on the Salary column, the database can use the index to quickly locate rows that meet the condition (Salary > 50000).
The database doesn't have to scan every row in the table. Instead, it performs an Index Range Scan.
The B-tree index on Salary allows the database to directly find the range of rows where Salary exceeds 50,000.