Documentation\_2

View

A ‘view’ in PostgreSQL is a virtual table that displays data based on a pre-defined SQL query, essentially acting as a filtered or calculated representation of data from one or more existing table, without actually storing the data separately, you can query a view just like you would in regular table.

**Benefits of using views:**

* By combining data from multiple tables into a single view, you can create cleaner and easier to understand queries for users.
* You can create views that only expose specific columns or filtered data based on access permissions.
* Views can be used to customize how data is displayed for different applications or reports.

**Points To Be Noted:**

**No data storage:**

Views don not store their own data; they only define a query that is executed when accessed.

**Dynamic updates:**

If the underlying tables are update, the data displays by the view will automatically reflect those changes.

**Access control:**

You can grant permissions to access a view just like you would for a normal table, allowing fine-grained control over data visibility.

**How to use the VEIW:**

To access the data in the view, simply query it like any other table.



**Example:**

Image you have a “customers” table with columns like “first\_name”, “last\_name”, and “city”, and you want to create a view that only shows customers from “New York” city:

CREATE VIEW new\_york\_customers AS

SELECT first\_name, last\_name, email

FROM customers

WHERE city = ‘New York’;

Materialized View

Materialized Views are database objects that store the result of a query physically and provide indirect access to real table data, significantly speeding up complex query retrieval times. Unlike standard views, they don't reflect the latest data in the underlying tables unless refreshed.

**Benefits of using Materialized views:**

* Since materialized views store the result of the query, queries against the materialized view are much faster than executing the same query repeatedly, especially for complex or resource-intensive queries.
* For queries that involve expensive joins, aggregations, or calculations, the materialized view provides a precomputed result, avoiding the need to recompute it each time the query is run.
* Materialized views can significantly reduce the load on the database server by precomputing and storing expensive computations. This can be useful in high-load environments, where executing complex queries repeatedly would put a strain on system resources.
* In data warehousing scenarios, materialized views can be used to preaggregate and store large amounts of summarized data, which makes subsequent queries run faster.

**Points To Be Noted:**

* Materialized views can be a powerful tool for improving query performance, especially for read-heavy workloads and complex aggregations.
* However, they come with trade-offs in terms of maintenance overhead, data freshness, and storage requirements.
* Proper planning for refresh strategies, indexing, and disk space management is essential to leverage the benefits while minimizing the drawbacks.
* Materialized views are ideal for scenarios where query speed is critical, and the data does not change frequently or can tolerate some delay in being updated.

**How to use the Materialized VEIW:**

To access the data in the Materialized view, simply query it like any other table.



**Example:**

CREATE MATERIALIZED VIEW monthly\_sales AS

SELECT EXTRACT (MONTH FROM sale\_date) AS month,

SUM (amount) AS total\_sales

FROM sales

GROUP BY month;

Common Table Expressions (CTEs)

A Common Table Expression (CTE) is a powerful feature in SQL that allows you to define a temporary result set which can be referenced within a SELECT, INSERT, UPDATE, or DELETE statement. CTEs can improve the readability and maintainability of your queries, especially when dealing with complex queries.

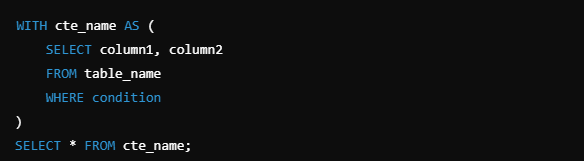
**Benefits of using CTE:**

* CTEs simplify complex queries by breaking them into smaller, logical components.
* Each CTE can represent a step in the query process, making it easier to understand.
* Once defined, a CTE can be referenced multiple times within the same query, avoiding redundant code.
* CTEs can encapsulate complex calculations, joins, or filtering logic, reducing clutter in the main query.

**Points To Be Noted:**

* A CTE exists only for the duration of the query in which it is defined. It cannot be reused in a different query unless redefined.
* A CTE is defined using the WITH keyword, followed by the CTE name and query.
* The CTE name must be unique within the query to avoid conflicts.
* Recursive CTEs must include an anchor member (non-recursive part) and a recursive member.
* They require the UNION ALL keyword.

**How to use CTE:**



To access the data in the CTE, simply query it like any other table.

**Example:**

SELECT name

FROM (SELECT name, salary FROM employees WHERE salary > 50000) AS subquery

WHERE name LIKE 'A%';