Views

Definition:

In PostgreSQL, a view is a virtual table that is based on the result of a query. It does not store data itself but provides a way to access data from one or more tables in a specific format. A view can be treated as a table in queries, but it dynamically generates the data every time it is accessed, based on the underlying query.

A view can simplify complex queries, improve security (by restricting access to certain columns or rows), and allow reuse of common queries.

Purpose of Views:

- **Simplification of Complex Queries**: Views allow us to encapsulate complex queries and provide a simpler interface to the user. Instead of repeatedly writing the same complex query, we can create a view and query the view like a regular table.
- **Data Security**: Views can be used to restrict access to sensitive data by only exposing certain columns or rows to the user. We can create a view that filters or aggregates data and grant access to the view rather than the underlying tables.
- **Consistency**: Views provide a consistent way of accessing data. Even if the structure of the underlying tables changes, the view can remain unchanged, providing a stable interface for applications.
- **Data Abstraction**: Views allow us to abstract the complexities of data schema or relationships, presenting a simplified and unified view of the data to users or applications.
- **Performance Optimization**: While views themselves don’t directly improve performance, they can be used to pre-define and simplify queries that would otherwise be repeated frequently, reducing the need for users to re-execute complex joins or aggregations.

Types of Views:

1. **Simple View**: A view that is based on a single query, usually retrieving data from one or more tables with no complex calculations or aggregations.
2. **Complex View:** A view based on a query that includes complex operations such as JOIN, GROUP BY, HAVING, or subqueries.
3. **Materialized View**: Unlike regular views, materialized views store the result of the query physically. They are used for performance optimization, as the data is precomputed and stored, reducing the cost of repetitive queries. Materialized views can be refreshed periodically to reflect changes in the underlying data.
4. **Updatable View**: A view that allows INSERT, UPDATE, or DELETE operations. However, not all views are updatable. A view is only updatable if the query underlying it does not contain elements that make it ambiguous or inconsistent (e.g., aggregate functions or joins that cannot be resolved directly).

Basic Syntax for Creating Views:

```
CREATE VIEW view_name AS
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

Parameters:

- view_name: The name of the view we want to create.
- SELECT column1, column2, ...: The query that defines the view's data. We can select columns from one or more tables.
- FROM table_name: The table(s) from which we are selecting the data.
- WHERE condition: Any condition to filter the rows returned by the query.

Example 1: Creating a Simple View

This example creates a view that shows the employee names and their salaries from the employees table.

```
CREATE VIEW employee_salary_view AS
SELECT name, salary
FROM employees;

-- Usage
SELECT * FROM employee_salary_view;
```

Explanation:

- The view employee_salary_view is created with the SELECT name, salary statement from the employees table.
- You can now query the view as if it were a table. SELECT * FROM employee_salary_view; will return the names and salaries of all employees.

Benefits:

- You no longer need to write the SELECT statement every time. Simply querying the view employee_salary_view is easier and cleaner.
- Views can also be used to abstract complex queries, making it easier for users to interact with the data.

Example 2: Creating a View with Joins

This example creates a view that shows employee names along with their department names by joining the employees and departments tables.

```
CREATE VIEW employee_department_view AS
SELECT e.id, e.name AS employee_name, d.department_name
FROM employees e
```

```
JOIN departments d ON e.department_id = d.id;

-- Usage
SELECT * FROM employee_department_view;
```

Explanation:

- The employee_department_view combines data from the employees and departments tables by performing an INNER JOIN on department_id.
- The resulting view shows each employee's ID, name, and department name.
- This view abstracts the complexity of the join and makes it easy for users to retrieve both employee and department information in a single query.

Example 3: Creating a View with Aggregation
This example creates a view that shows the total salary for each department in the employees table.

```
CREATE VIEW department_salary_summary AS
SELECT department_id, SUM(salary) AS total_salary
FROM employees
GROUP BY department_id;

-- Usage
SELECT * FROM department_salary_summary;
```

Explanation:

- The view department_salary_summary calculates the total salary (SUM(salary)) for each department (department_id) using the GROUP BY clause.
- The resulting view gives a summary of salaries for each department.

Benefits:

- Aggregation views help simplify complex data analysis queries, providing a ready-to-use summary without needing to re-write the aggregation logic every time.

Example 4: Creating a View with a WHERE Clause
This example creates a view that only shows employees who earn more than $50,000.

```
CREATE VIEW high_salary_employees AS
SELECT name, salary
FROM employees
WHERE salary > 50000;

-- Usage
SELECT * FROM high_salary_employees;
```

Explanation:

- The view high_salary_employees filters the employees table to only include those who have a salary greater than $50,000.
- This view simplifies querying for high-paid employees and abstracts the filtering logic for users.

Example 5: Creating an Updatable View
This example creates an updatable view that allows us to easily update an employee's salary.

```
CREATE VIEW employee_salary_view AS
SELECT id, name, salary
FROM employees;

-- Usage: You can now update the salary through the view
UPDATE employee_salary_view
SET salary = 60000
WHERE name = 'Alice';

-- Verifying the update
SELECT * FROM employees WHERE name = 'Alice';  -- The salary for Alice should now be
60000
```

Explanation:

- The view employee_salary_view is based on the employees table and includes the employee's id, name, and salary.
- Since the view only selects from a single table (employees) and doesn't include any complex joins, you can perform updates directly on it.
- The UPDATE operation on the view updates the underlying table (employees).

Note: Not all views are updatable. Views that involve JOINs, GROUP BY, or DISTINCT are typically not updatable.

Example 6: Creating and Using a Materialized View
This example creates a materialized view that stores the total salary by department, and you can refresh the view periodically to keep it up to date.

```
-- Create a materialized view
CREATE MATERIALIZED VIEW department_salary_summary_matview AS
SELECT department_id, SUM(salary) AS total_salary
FROM employees
GROUP BY department_id;

-- Usage
SELECT * FROM department_salary_summary_matview;
```

```
-- Refreshing the materialized view when the underlying data changes
REFRESH MATERIALIZED VIEW department_salary_summary_matview;
```

Explanation:

- The department_salary_summary_matview materialized view calculates the total salary by department.
- Once created, the data is physically stored in the materialized view, making subsequent queries faster.
- When the underlying data in the employees table changes, you must explicitly refresh the materialized view with the REFRESH MATERIALIZED VIEW statement to update its data.

Benefits of Materialized Views:

- They are useful for complex queries where you want to cache the results for fast retrieval.
- You can control when to refresh the data, which can be useful in reporting scenarios where real-time data is not necessary.

Example 7: Dropping a View

```
DROP VIEW IF EXISTS employee_salary_view;
```

Explanation:

- The DROP VIEW statement removes the view from the database.
- The IF EXISTS clause ensures that no error is thrown if the view doesn’t exist.