SQL Views: A Detailed Explanation

What is a View?

A view in SQL is a virtual table that represents the result of a SQL query. It is a stored query that users can query like a regular table. However, it does not store data itself; it displays data fetched dynamically from the underlying tables whenever accessed.

Think of a view as a saved query: instead of writing the same query over and over again, you can create a view and just query the view.

Why do we need Views?

1. Data Security: Views can be used to restrict access to specific rows or columns in a table. For instance, if some users should not see salary data in an employee table, you can create a view that excludes the salary column.

2. Simplifying Queries: Complex SQL queries involving multiple joins or calculations can be simplified by encapsulating them in views. This simplifies the SQL code that users or applications need to run.

3. Data Abstraction: Views can provide a customized presentation of the data. Users can see data in a specific format without worrying about the complexity of the underlying tables.

4. Reuse of Queries: Once a view is created, it can be reused multiple times without needing to re-write the complex logic.

How to Create a View

A view is created using the CREATE VIEW statement followed by the SQL query that defines the view.

Syntax:

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

Real-Life Example:

Imagine we have an employees table with the following structure:

We want to create a view that hides the salary column from general users and only displays the name and department.

Step-by-Step Creation of a View:

```
CREATE VIEW view_employee AS
SELECT name, department
FROM employees;
```

1. CREATE VIEW view_employee: This part creates a view called view_employee. A view name should follow the same naming conventions as table names.

2. SELECT name, department: We specify the columns (name and department) that should appear in the view.

3. FROM employees: This indicates that the data is sourced from the employees table.

Now, when we query the view view_employee:

```
SELECT * FROM view_employee;
```

The output will be:

Here, the salary column is hidden, fulfilling the security requirement.

Advanced Example:

Suppose we want to show only IT employees in a view:

```
CREATE VIEW view_IT_employees AS
SELECT name, salary
FROM employees
WHERE department = 'IT';
```

This view will filter only IT employees and show their names and salaries.

Advantages of Views:

1. Security: You can limit access to certain rows and columns. For example, if you want some users to see only non-sensitive data, views can enforce that.

2. Simplification of Complex Queries: Users don't need to write long and complex queries repeatedly. They can simply use views to get the same data.

3. Data Abstraction: Views can hide the complexity of the database schema. Users can interact with a simpler, logical representation of the data.

4. Consistency: Views provide a consistent interface to query the data, even if the underlying tables change. As long as the view definition remains valid, users won't need to change their queries.

Disadvantages of Views:

1. Performance: Since a view doesn't store data physically, every time you query a view, the underlying query is executed. If the query is complex, this can impact performance.

2. Dependency: If the structure of the underlying table changes (e.g., if a column used in the view is removed), the view may become invalid.

3. Limited Insert/Update/Delete: Depending on the complexity of the view (e.g., if it joins multiple tables or uses certain SQL functions), updating the underlying data through the view can be restricted.

4. Not Always Indexable: Views may not benefit from indexes in the same way that tables do, especially when working with complex queries.

Modifying Views:

If we want to modify an existing view, we use the CREATE OR REPLACE VIEW statement.

For example, if we want to update our view_IT_employees to include department information:

```
CREATE OR REPLACE VIEW view_IT_employees AS
SELECT name, department, salary
FROM employees
WHERE department = 'IT';
```

This modifies the existing view to include the department column.

Dropping a View:

If you no longer need a view, you can drop it using:

DROP VIEW view_name;

For example:

DROP VIEW view_IT_employees;
.

More Examples with SQL Views

Let’s expand on the basic view creation with more operations like UPDATE, INSERT, and DELETE. In many cases, updates to a view can be restricted or behave differently than when you update a regular table, especially if the view involves complex queries like joins, aggregations, or subqueries. However, if a view directly maps to a single table, certain operations may still work.

1. Updating Data through a View

If your view maps directly to a single table and doesn’t include complex operations like joins or aggregates, you can often perform UPDATE operations on the view, which will reflect in the underlying table.

Example:

Let’s use the employees table from before:

We create a simple view for the employees working in the IT department:

```
CREATE VIEW view_IT_employees AS
SELECT emp_id, name, salary
FROM employees
WHERE department = 'IT';
```

Now, let’s say we want to give Bob a raise. We can update Bob’s salary through the view.

```
UPDATE view_IT_employees
SET salary = 5000
WHERE name = 'Bob';
```

UPDATE view_IT_employees: This specifies that we want to update data in the view.

SET salary = 5000: We are setting the salary to 5000 for the employee named Bob.

WHERE name = 'Bob': This clause filters to only the row where the employee’s name is Bob.

After this operation, the underlying employees table will reflect the change:

Note: You can only update data through a view if:

The view includes all columns necessary for the UPDATE.

The view is based on a single table and does not involve aggregations, groupings, or complex joins.

2. Inserting Data through a View

Inserting data into a view can be tricky and is only allowed in certain situations. The view must have a one-to-one correspondence with the underlying table (i.e., no joins or aggregate functions), and it must include all columns that are required (e.g., primary key columns).

Let's take the view view_IT_employees again. We can insert a new IT employee into the employees table through the view:

```
INSERT INTO view_IT_employees (emp_id, name, salary)
VALUES (5, 'David', 4800);
```

INSERT INTO view_IT_employees: We are inserting a new record into the view.

(emp_id, name, salary): We specify the columns we want to insert data into. Since the view doesn't include the department column, we can't insert the department directly.

VALUES (5, 'David', 4800): These are the values for the new record.

After this, the underlying table will be updated with the new employee:

Since the view doesn't show the department column, the view will automatically assign the department value of 'IT' based on the filter used when defining the view (WHERE department = 'IT').

3. Deleting Data through a View

Just like with UPDATE and INSERT, you can delete rows through a view as long as it's a simple view with a direct mapping to the underlying table.

Let's say we want to remove the employee named 'David':

```
DELETE FROM view_IT_employees
WHERE name = 'David';
```

DELETE FROM view_IT_employees: This tells the database to delete rows from the view.

WHERE name = 'David': The filter ensures we are deleting only the row where the employee's name is David.

After this, David's record will be removed from the employees table:

Note: Similar to the other operations, deleting through a view is only allowed when:

The view is based on a single table.

The view doesn't include aggregated columns or complex joins.

4. Views with Joins

Views can be used to simplify complex queries involving joins. For example, let's say we have two tables: employees and departments.

employees table:

departments table:

We can create a view that joins the employees table with the departments table to show department names instead of department_id:

```
CREATE VIEW view_employee_departments AS
SELECT e.emp_id, e.name, d.department_name, e.salary
FROM employees e
JOIN departments d
ON e.department_id = d.department_id;
```

JOIN departments d ON e.department_id = d.department_id: This joins the employees and departments tables based on the matching department_id.

SELECT e.emp_id, e.name, d.department_name, e.salary: The view now displays the employee's name, department name (from the departments table), and salary.

When you query the view_employee_departments view:

```
SELECT * FROM view_employee_departments;
```

The result would be:

This simplifies complex joins, allowing users to interact with the data without needing to understand the underlying table relationships.

Limitations of Updating Views with Joins:

While views with joins simplify data presentation, you cannot directly update or insert into such views because there is ambiguity about which table the data should be inserted into or updated.

For example, you cannot update the department_name through the view because it comes from the departments table, and it's unclear whether you want to change the employee's department or modify the department itself.

Security through SQL Views vs. Regular Queries with WHERE

Using views for security is a more structured and maintainable approach than simply relying on WHERE clauses in individual queries. Here's a detailed explanation of how views enhance security and how they differ from directly applying WHERE clauses in standard queries.

1. Centralized Control over Data Access:

When using a view, you define a single point of control for data access. This ensures that everyone using the view sees only the data they are allowed to access, without having to rely on them remembering to add a WHERE clause.

Example:

Assume you have an employees table that contains sensitive salary data, and you want to restrict some users from seeing the salary column.

Without a View: You would need to rely on users or application developers to always include the correct WHERE clause to restrict access to the salary column in their queries. This leaves room for error, as any user who forgets to add the restriction may access the sensitive data.

```
SELECT name, department
FROM employees
WHERE department = 'IT';
```

While this query omits the salary data for IT employees, there's no guarantee that future queries or users will remember to leave out the salary column.

With a View: You can define a view that permanently excludes the salary data for all users who access it:

```
CREATE VIEW view_non_sensitive_employees AS
SELECT name, department
FROM employees;
```

This way, whenever users access view_non_sensitive_employees, they cannot access the salary column, regardless of what query they run:

SELECT * FROM view_non_sensitive_employees;

The salary data is guaranteed to be inaccessible.

2. Role-Based Access Control (RBAC):

Views allow fine-grained access control by assigning different views to different user roles. Instead of managing access at the query level, you can restrict access to certain views for certain users.

Example:

If you have two roles in your organization:

HR: Needs access to salary information.

General Employees: Should not see salary information.

You can create two different views:

1. For HR (with salary):

```
CREATE VIEW view_hr_access AS
SELECT emp_id, name, department, salary
FROM employees;
```

2. For General Employees (without salary):

```
CREATE VIEW view_general_access AS
SELECT emp_id, name, department
FROM employees;
```

By giving each role access only to the appropriate view, you ensure that data is restricted based on roles without relying on the user to remember or apply the correct filters.

In contrast, applying a WHERE clause in a query can easily be bypassed or forgotten, potentially exposing sensitive information.

3. Data Masking with Views:

Views can be used to mask sensitive data in a consistent manner. This allows you to provide partial access to sensitive data without exposing the full details.

Example:

Consider an employees table with sensitive data like phone numbers or salaries. You can create a view that masks the salary, only showing a generic value (like ‘REDACTED’) to users who shouldn’t see the actual numbers:

```
CREATE VIEW view_masked_salary AS
SELECT emp_id, name, department,
      CASE
         WHEN department = 'HR' THEN salary
         ELSE 'REDACTED'
      END AS salary
FROM employees;
```

In this case, only HR employees will see the actual salary, while others will see ‘REDACTED’. This kind of data masking is hard to implement effectively with just WHERE clauses, as it requires specific logic to be applied consistently across queries.

When users query this view:

```
SELECT * FROM view_masked_salary;
```

The output for non-HR users would look like this:

This provides a more flexible and secure way to control data exposure.

4. Reduced Attack Surface:

Relying on WHERE clauses for security means that the burden is on the query author (or the application) to apply the correct filters. If someone makes a mistake or tries to bypass the filter, they might gain unauthorized access to data.

Views reduce this attack surface by predefining what data can be accessed and preventing users from querying other data without administrative permissions.

Example:

An application might contain several SQL queries across different components. Ensuring that every single query includes the correct WHERE clause is error-prone and difficult to audit. If a developer forgets to apply the clause, users might see more data than they should.

By creating views, you can lock down data access at the database level. The view itself ensures that only the appropriate data is visible, regardless of what query is run. Users cannot accidentally or deliberately bypass these restrictions.

5. Consistency in Data Access:

When using WHERE clauses manually in each query, there is always the possibility that different users or applications may write inconsistent queries, leading to inconsistent data access or exposure.

Views provide consistent access to the same data set for all users or applications. You define the logic once in the view, and all users who query that view will see the same results, ensuring uniformity in data access.

Example:

If one application queries an employee table with this:

```
SELECT name, department
FROM employees
WHERE department = 'IT';
```

And another application queries with a slightly different filter:

```
SELECT name, department
FROM employees
WHERE department IN ('IT', 'HR');
```

The data being accessed is inconsistent. Views prevent such discrepancies by centralizing the filtering logic.

6. Views Can be Used for Permissions Control:

In many database systems (like PostgreSQL, MySQL, or SQL Server), you can control which users have access to a view. For example, you can revoke access to the underlying table and give users access only to the view. This prevents them from running any arbitrary query on the full table.

Example:

```
REVOKE ALL ON employees FROM general_user;
GRANT SELECT ON view_non_sensitive_employees TO general_user;
```

In this case, the general_user cannot query the employees table directly, but they can access the view_non_sensitive_employees, which ensures they only see non-sensitive data.

7. Ease of Maintenance:

Let’s say you later decide that the data in the employees table should be filtered differently. If you are using WHERE clauses across various queries, you’d need to update every instance where this filter is applied. However, if you use a view, you can simply update the view definition, and the change will automatically propagate to all users or applications that query the view.

Example:

Imagine you change the policy and now only IT employees with a salary greater than 4500 should be visible:

```
CREATE OR REPLACE VIEW view_IT_employees AS
SELECT name, salary
FROM employees
WHERE department = 'IT' AND salary > 4500;
```

Now all queries using view_IT_employees will respect the updated filter, without needing to update each individual query.

Summary of Differences:

While both views and WHERE clauses can restrict data access, views provide a centralized, consistent, and more secure mechanism for controlling what data is visible. By using views, database administrators can enforce security policies at the database level, reducing the risk of human error or intentional circumvention that might occur when relying on WHERE clauses in individual queries.

Conclusion:

SQL views are a powerful tool for abstracting, simplifying, and securing access to data. While simple views allow operations like UPDATE, INSERT, and DELETE, more complex views (involving joins, groupings, or aggregations) may impose restrictions. Understanding these limitations is key when using views in real-life scenarios to ensure efficiency and accuracy.