**a View is a virtual table that consists of a stored query based on a SELECT query . It does not store data itself but dynamically retrieves data from one or more underlying tables whenever the view is queried. Views are useful for simplifying complex queries, providing an abstracted layer for users, and ensuring data security by restricting access to sensitive data.**

**the view dynamically fetches the latest data from the table each time you query it.**

**Advantages and Disadvantages of Views**

|  |  |  |
| --- | --- | --- |
| Aspect | Without View | With View |
| Data Access | Access to the entire table; requires more filtering. | Simplifies data access by abstracting complex queries. |
| Complexity | Complex queries need to be written repeatedly. | Encapsulates complex queries for reuse. |
| Data Security | Exposes entire table; sensitive data must be filtered manually. | Restricts access to specific columns/rows in the View. |
| Performance | May require optimization for each query. | Can improve query performance with precompiled views, but may not for large datasets. |
| Flexibility | Offers full flexibility in querying. | Limited by the structure of the View. |
| Storage | No additional storage required. | Views are virtual (no storage), except materialized views. |
| Maintenance | Complex queries in multiple places can increase maintenance. | Centralizes query logic, making maintenance easier. |
| Dependency | No dependency issues. | Changes to base tables or schema can affect the View. |

**Operations While Creating and After Creating Views**

|  |  |  |
| --- | --- | --- |
| Operation | During View Creation | After View Creation |
| SELECT Statement | Can use any valid SELECT query. | Can query the View like a table. |
| ORDER BY Clause | Not allowed directly in the CREATE VIEW. | Can use ORDER BY in queries on the View. |
| WHERE Clause | Allowed to filter rows in the View. | Can further filter rows in queries. |
| Aggregations | Allowed (SUM, COUNT, etc.). | Aggregations can be performed on the View. |
| Joins | Can include joins. | Queries can join the View with other tables or views. |
| Insert/Update/Delete | Limited to certain conditions (e.g., no joins, distinct, etc.). | Depends on whether the View is updatable. |
| Indexes | Cannot define indexes on Views directly. | Not applicable unless it’s a materialized View. |
| Modifying Data | Cannot modify data directly in the View if it’s non-updatable. | Requires altering the base table. |
| Schema Changes | Reflects changes in the underlying table. | May fail if schema changes break the View definition. |

**Key Rules for SQL Views**

1. **During Creation**:
   * Views are defined using a CREATE VIEW statement.
   * Cannot include an ORDER BY clause directly.
   * Aggregations, joins, and subqueries are allowed.

**Example**:

CREATE VIEW EmployeeSummary AS

SELECT DepartmentID, AVG(Salary) AS AvgSalary

FROM Employees

GROUP BY DepartmentID;

1. **After Creation**:
   * Queries on Views can include an ORDER BY clause.
   * Views behave like virtual tables and can be queried using SELECT.
   * Updates to a View are possible only if it’s **updatable**:
     + The View must directly map to one table.
     + No DISTINCT, GROUP BY, HAVING, or set operations in the query.

**Example**:

SELECT \* FROM EmployeeSummary ORDER BY AvgSalary DESC;

**View Limitations**

* **Non-Updatable Views**:
  + Views with joins, aggregations, or DISTINCT cannot be directly updated.
* **Performance**:
  + Views may not improve performance for every query, as they are recalculated unless materialized.
* **Schema Dependency**:
  + Changes to the base table schema can invalidate the View.

**Example: Working with Views in SQL**

**Step 1: Create a Base Table**

We’ll create a table named Employees for demonstration purposes.

CREATE TABLE Employees (

EmployeeID INT PRIMARY KEY,

Name VARCHAR(50),

DepartmentID INT,

Salary DECIMAL(10, 2),

HireDate DATE

);

INSERT INTO Employees (EmployeeID, Name, DepartmentID, Salary, HireDate) VALUES

(1, 'Alice', 101, 60000, '2020-01-15'),

(2, 'Bob', 102, 75000, '2019-03-10'),

(3, 'Charlie', 101, 50000, '2021-07-01'),

(4, 'David', 103, 90000, '2018-11-23'),

(5, 'Eve', 102, 72000, '2022-05-15');

**Step 2: Creating a View**

We’ll create a View named EmployeeSummary that summarizes employee data.

CREATE VIEW EmployeeSummary AS

SELECT DepartmentID, COUNT(EmployeeID) AS TotalEmployees, AVG(Salary) AS AvgSalary

FROM Employees

GROUP BY DepartmentID;

**Step 3: Querying the View After Creation**

You can perform operations on the created View.

**1. Selecting Data from the View**

SELECT \* FROM EmployeeSummary;

**Result**:

| **DepartmentID** | **TotalEmployees** | **AvgSalary** |
| --- | --- | --- |
| 101 | 2 | 55000.00 |
| 102 | 2 | 73500.00 |
| 103 | 1 | 90000.00 |

**2. Filtering Data Using WHERE**

SELECT \* FROM EmployeeSummary WHERE AvgSalary > 60000;

**Result**:

| **DepartmentID** | **TotalEmployees** | **AvgSalary** |
| --- | --- | --- |
| 102 | 2 | 73500.00 |
| 103 | 1 | 90000.00 |

**3. Ordering Data Using ORDER BY**

SELECT \* FROM EmployeeSummary ORDER BY AvgSalary DESC;

**Result**:

| **DepartmentID** | **TotalEmployees** | **AvgSalary** |
| --- | --- | --- |
| 103 | 1 | 90000.00 |
| 102 | 2 | 73500.00 |
| 101 | 2 | 55000.00 |

**Step 4: Limitations While Creating the View**

**1. Using ORDER BY in the View Definition (Not Allowed)**

CREATE VIEW OrderedEmployees AS

SELECT Name, Salary

FROM Employees

ORDER BY Salary DESC; -- This will throw an error.

**2. Using WHERE and Aggregations**

Filters and aggregations are allowed during View creation.

CREATE VIEW HighEarners AS

SELECT Name, Salary

FROM Employees

WHERE Salary > 70000;

Querying the View:

SELECT \* FROM HighEarners;

**Result**:

| **Name** | **Salary** |
| --- | --- |
| Bob | 75000.00 |
| David | 90000.00 |
| Eve | 72000.00 |

**Step 5: Performing Updates on Views**

**Updatable View**

Create a View for direct mapping to a single table:

CREATE VIEW SimpleEmployees AS

SELECT EmployeeID, Name, Salary

FROM Employees;

Update using the View:

UPDATE SimpleEmployees

SET Salary = 80000

WHERE Name = 'Alice';

Querying the base table:

SELECT \* FROM Employees;

**Result**:

| **EmployeeID** | **Name** | **DepartmentID** | **Salary** | **HireDate** |
| --- | --- | --- | --- | --- |
| 1 | Alice | 101 | 80000.00 | 2020-01-15 |
| 2 | Bob | 102 | 75000.00 | 2019-03-10 |
| 3 | Charlie | 101 | 50000.00 | 2021-07-01 |
| 4 | David | 103 | 90000.00 | 2018-11-23 |
| 5 | Eve | 102 | 72000.00 | 2022-05-15 |

**Non-Updatable View**

If a View contains aggregations, joins, or DISTINCT, it becomes non-updatable.

Example:

UPDATE EmployeeSummary

SET AvgSalary = 60000; -- This will throw an error as the View is not updatable.

**Summary**

|  |  |  |
| --- | --- | --- |
| Operation | Allowed During Creation? | Allowed After Creation? |
| SELECT | Yes | Yes |
| WHERE | Yes | Yes |
| ORDER BY | No | Yes |
| Aggregations (SUM, AVG) | Yes | Yes |
| Joins | Yes | Yes |
| Insert/Update/Delete | Yes (in certain conditions) | Yes (only if updatable) |
|  |  |  |

**Operations While Creating Views and After Creating Views**

**While Creating Views**

When creating a view, certain SQL operations and clauses are allowed or restricted. Views are virtual tables based on a SQL query and serve to simplify queries or provide data security.

|  |  |  |
| --- | --- | --- |
| Operation/Clause | Allowed During View Creation? | Example |
| SELECT | Yes | sql CREATE VIEW ViewName AS SELECT \* FROM TableName; |
| WHERE | Yes | sql CREATE VIEW ActiveEmployees AS SELECT \* FROM Employees WHERE Status = 'Active'; |
| JOIN | Yes | sql CREATE VIEW EmployeeDetails AS SELECT e.Name, d.DeptName FROM Employees e JOIN Departments d ON e.DeptID = d.DeptID; |
| GROUP BY | Yes | sql CREATE VIEW AvgSalaryByDept AS SELECT DeptID, AVG(Salary) AS AvgSalary FROM Employees GROUP BY DeptID; |
| HAVING | Yes | sql CREATE VIEW HighAvgSalaryDepts AS SELECT DeptID, AVG(Salary) AS AvgSalary FROM Employees GROUP BY DeptID HAVING AVG(Salary) > 60000; |
| ORDER BY | Yes | sql CREATE VIEW OrderedEmployees AS SELECT \* FROM Employees ORDER BY Salary DESC; |
| DISTINCT | Yes | sql CREATE VIEW UniqueDepartments AS SELECT DISTINCT DeptName FROM Departments; |
| SET Operations (UNION) | Yes | sql CREATE VIEW AllEmployees AS SELECT Name FROM Employees\_2023 UNION SELECT Name FROM Employees\_2022; |
| INSERT/UPDATE/DELETE | No | You cannot perform these during view creation. Views are defined with CREATE VIEW. |

These operations can only be performed **after the view is created** and only if the view is **updatable**.

**Why INSERT/UPDATE/DELETE Cannot Be Performed During View Creation?**

When creating a view using the CREATE VIEW statement, the purpose is to define a virtual table based on a SQL query. Operations like INSERT, UPDATE, or DELETE modify data in tables, which is not the function of view creation. These operations can only be performed **after the view is created** and only if the view is **updatable**.

**Example**

**Base Table: Employees**

CREATE TABLE Employees (

EmployeeID INT PRIMARY KEY,

Name VARCHAR(50),

DepartmentID INT,

Salary DECIMAL(10, 2)

);

INSERT INTO Employees (EmployeeID, Name, DepartmentID, Salary) VALUES

(1, 'Alice', 101, 60000),

(2, 'Bob', 102, 75000);

**Attempt to Use INSERT/UPDATE/DELETE During View Creation**

CREATE VIEW ActiveEmployees AS

INSERT INTO Employees (EmployeeID, Name, DepartmentID, Salary)

VALUES (3, 'Charlie', 101, 50000);

**Result:** This will throw a syntax error because INSERT is not allowed in the CREATE VIEW statement.

**Explanation**

The CREATE VIEW statement is only for defining the structure of the view. The query provided in the view definition is used to populate the view **dynamically** whenever it is queried. Modifying data (INSERT/UPDATE/DELETE) is a separate operation and cannot be part of the view creation process.

**Correct Way: Create View First, Then Perform Operations**

1. **Create the View:**

CREATE VIEW HighSalaryEmployees AS

SELECT \* FROM Employees WHERE Salary > 70000;

1. **Perform INSERT into View (if updatable):**

INSERT INTO Employees (EmployeeID, Name, DepartmentID, Salary)

VALUES (3, 'Charlie', 102, 80000);

Note: Data modification (INSERT/UPDATE/DELETE) happens directly on the base table, not on the view creation statement.

1. **Query the View to Reflect Changes:**

SELECT \* FROM HighSalaryEmployees;

**Key Takeaway**

The CREATE VIEW statement is solely for defining the structure and logic of the view. Operations that modify data like INSERT, UPDATE, or DELETE cannot be performed during view creation—they must be executed separately after the view is created.

**After Creating Views**

Once a view is created, you can perform queries and sometimes updates on it. The allowed operations depend on whether the view is **updatable** or **non-updatable**.

|  |  |  |
| --- | --- | --- |
| Operation | Allowed? | Example |
| SELECT Query | Yes | sql SELECT \* FROM ViewName; |
| WHERE Clause | Yes | sql SELECT \* FROM ViewName WHERE Salary > 70000; |
| ORDER BY Clause | Yes | sql SELECT \* FROM ViewName ORDER BY Name ASC; |
| INSERT into View | Only for Updatable Views | sql INSERT INTO ViewName (columns...) VALUES (...); |
| UPDATE View | Only for Updatable Views | sql UPDATE ViewName SET ColumnName = 'NewValue' WHERE Condition; |
| DELETE from View | Only for Updatable Views | sql DELETE FROM ViewName WHERE Condition; |
| DML on Non-Updatable View | No | Non-updatable views will not allow INSERT, UPDATE, or DELETE. |

**Examples**

**1. Creating a View**

CREATE VIEW ActiveEmployees AS

SELECT EmployeeID, Name, Salary

FROM Employees

WHERE Status = 'Active';

**2. Querying a View**

SELECT \* FROM ActiveEmployees;

**Output:**

| **EmployeeID** | **Name** | **Salary** |
| --- | --- | --- |
| 1 | Alice | 70000 |
| 2 | Bob | 80000 |

**3. Updating a View (Updatable View)**

If the view is based directly on one table and satisfies the **conditions for updatability**, you can update it.

**Example:**

UPDATE ActiveEmployees

SET Salary = 75000

WHERE EmployeeID = 1;

**Effect:**

* The **base table** (Employees) is updated.
* The change is reflected in the view.

**4. Inserting into a View (Updatable View)**

INSERT INTO ActiveEmployees (EmployeeID, Name, Salary)

VALUES (3, 'Charlie', 60000);

**Effect:**

* The row is inserted into the base table (Employees).
* The view is updated if the condition (Status = 'Active') is satisfied.

**5. Updating Non-Updatable Views**

Non-updatable views use aggregations, joins, or other operations like DISTINCT, GROUP BY, or HAVING. These make direct updates impossible.

**Example:**

CREATE VIEW AvgSalaryByDept AS

SELECT DeptID, AVG(Salary) AS AvgSalary

FROM Employees

GROUP BY DeptID;

**Attempt to Update:**

UPDATE AvgSalaryByDept

SET AvgSalary = 70000

WHERE DeptID = 101;

**Result:** Error, because SQL cannot map the aggregated value (AvgSalary) back to individual rows in the base table.

**6. Dropping a View**

DROP VIEW ActiveEmployees;

**Key Points for Updatability**

A view is **updatable** only if:

1. It directly maps to a single table.
2. It doesn’t use clauses like DISTINCT, GROUP BY, HAVING, UNION, or aggregate functions.
3. It doesn’t include calculated columns or joins.

**Basic Concepts**

1. **What is a View in SQL?**  
   A. A temporary table that stores data  
   B. A virtual table based on a SELECT query  
   C. A physical table that stores a subset of columns  
   D. A system-defined table that stores metadata

**Answer:** B

1. **Which SQL command is used to create a view?**  
   A. CREATE TABLE  
   B. CREATE VIEW  
   C. SELECT INTO  
   D. ALTER VIEW

**Answer:** B

1. **Can a view be based on another view?**  
   A. Yes  
   B. No

**Answer:** A

1. **Which of the following is true about views?**  
   A. Views store data physically.  
   B. Views can be indexed.  
   C. Views can be created using the CREATE VIEW statement.  
   D. Views cannot include WHERE clauses.

**Answer:** C

**Advanced Concepts**

1. **Which of the following operations can be performed on a view if it is updatable?**  
   A. INSERT  
   B. UPDATE  
   C. DELETE  
   D. All of the above

**Answer:** D

1. **What will happen if the base table is deleted but the view still exists?**  
   A. The view will still work.  
   B. The view will be automatically deleted.  
   C. Queries on the view will result in an error.  
   D. The view will store the last known data.

**Answer:** C

1. **Which of the following conditions makes a view non-updatable?**  
   A. The view is based on multiple tables joined together.  
   B. The view contains aggregate functions.  
   C. The view has a DISTINCT keyword.  
   D. All of the above.

**Answer:** D

1. **Which clause is mandatory when creating a view?**  
   A. WHERE  
   B. GROUP BY  
   C. SELECT  
   D. ORDER BY

**Answer:** C

**Practical Scenarios**

1. **If a view is created with the query SELECT \* FROM Employees WHERE Department = 'HR';, which of the following is true?**  
   A. The view will always include all employees.  
   B. The view will dynamically show only HR employees.  
   C. You cannot use this view to insert new employees.  
   D. Both B and C.

**Answer:** D

Inserting data into a view is restricted if the view includes conditions like WHERE. Any INSERT operation on such a view is ambiguous because the database cannot guarantee that the inserted data will meet the WHERE condition.

**Restrictions on INSERT (C):** You cannot directly insert new rows into the HREmployees view because the WHERE Department = 'HR' condition complicates the INSERT operation. The database cannot ensure that the INSERT operation will satisfy the condition of the view.

1. **Which statement can be used to delete a view?**  
   A. DROP VIEW  
   B. DELETE VIEW  
   C. ALTER VIEW  
   D. REMOVE VIEW

**Answer:** A

1. **Can an ORDER BY clause be used in the CREATE VIEW statement?**  
   A. Yes, always.  
   B. No, never.  
   C. Yes, but only when combined with TOP or LIMIT.  
   D. Yes, but only if the database supports it.

**Answer:** C

**When You Can Use ORDER BY:**

* **With TOP or LIMIT**:  
  When you use TOP (SQL Server) or LIMIT (MySQL, PostgreSQL), you are selecting a specific subset of rows, and the order of those rows becomes relevant. Therefore, it is possible to use ORDER BY with these clauses.

**Example (SQL Server with TOP):**

CREATE VIEW TopEmployees AS

SELECT TOP 10 \* FROM Employees

ORDER BY Salary DESC;

**Example (MySQL/PostgreSQL with LIMIT):**

CREATE VIEW TopEmployees AS

SELECT \* FROM Employees

ORDER BY Salary DESC

LIMIT 10;

* **Without TOP/LIMIT**:  
  If you do not use TOP or LIMIT, then using ORDER BY in a CREATE VIEW statement typically **does not make sense** because a view is not intended to store data in a particular order. Views are queries that can be reused, and each time the view is queried, the result can be ordered as required.

1. **Which of the following is a benefit of using views?**  
   A. Improved query performance.  
   B. Data security by restricting column access.  
   C. Automatic updates to view when base table changes.  
   D. Both B and C.

**Answer:** D

**True/False Questions**

1. **Views can include aggregate functions like SUM and AVG.**  
   A. True  
   B. False

**Answer:** A

1. **Views can be used to simplify complex queries.**  
   A. True  
   B. False

**Answer:** A

1. **Views cannot accept parameters.**  
   A. True  
   B. False

**Answer:** A

you cannot do this:

CREATE VIEW EmployeesInDepartment AS

SELECT \* FROM Employees WHERE Department = @DeptName;

A materialized view stores the results of a query physically in the database, much like a snapshot of the data at the time of creation or last refresh. This means that querying a materialized view is faster compared to a regular view, because it avoids having to execute the underlying query every time.

Materialized views are **not automatically updated** when the underlying data changes (unless explicitly refreshed). You need to periodically **refresh** the materialized view to ensure it reflects the most recent data. This can be done manually or through an automatic refresh mechanism (depending on the database system)

CREATE MATERIALIZED VIEW SalesSummary AS

SELECT Region, SUM(SalesAmount) AS TotalSales

FROM Sales

GROUP BY Region;

REFRESH MATERIALIZED VIEW SalesSummary;

**Example Use Case:**

Imagine you have a large sales database with millions of records. You frequently run queries to get the total sales per region. Instead of running the same complex aggregation query every time, you can create a materialized view to store the aggregated results.

1. **Create the Materialized View:**

CREATE MATERIALIZED VIEW RegionalSales AS

SELECT Region, SUM(SalesAmount) AS TotalSales

FROM Sales

GROUP BY Region;

1. **Query the Materialized View:**

SELECT \* FROM RegionalSales;

The query will run much faster because the data is precomputed and stored in the materialized view.

1. **Refresh the Materialized View:** To ensure that the RegionalSales view reflects the latest data, you would refresh it periodically:

REFRESH MATERIALIZED VIEW RegionalSales;

**Advantages of Materialized Views:**

* **Faster Query Performance:** Precomputed results reduce the need for repeated complex calculations.
* **Improved Efficiency:** Helps reduce database load by storing results of expensive operations.
* **Can Be Refreshed:** You can control when the view is updated to reflect the most recent data.

**Disadvantages of Materialized Views:**

* **Storage Cost:** Materialized views consume additional storage since they store the query results physically.
* **Maintenance Overhead:** You need to manage the refresh process to keep the materialized view up-to-date. If you don't refresh it regularly, the data in the view may become outdated.

**In Summary:**

A **materialized view** is a precomputed, stored version of the result of a query. It helps improve performance by saving the query results and can be refreshed periodically to keep the data up-to-date. Unlike regular views, materialized views store the data physically, making them useful for scenarios where query performance is critical, but they require storage and maintenance to keep the data current.

8. Consider the two relations instructor and department  
Instructor:

|  |  |  |  |
| --- | --- | --- | --- |
| ID | Name | Dept\_name | Salary |
| 1001 | Ted | Finance | 10000 |
| 1002 | Bob | Music | 20000 |
| 1003 | Ron | Physics | 50000 |

Department:

|  |  |  |
| --- | --- | --- |
| Dept\_name | Building | Budget |
| Biology | Watson | 40000 |
| Chemistry | Painter | 30000 |
| Music | Taylor | 50000 |

Which of the following is used to create view for these relations together?  
a)

**CREATE** **VIEW** instructor\_info **AS**

**SELECT** ID, name, building

**FROM** instructor, department

**WHERE** instructor.dept name= department.dept name;

-------------------------------------------------------------------------------------------------------------------

In SQL, whether you can **insert** or **delete** rows in a **view** depends on several factors. Here's a breakdown:

**1. Inserting into a View:**

* **Updatable Views:** If the view is **updatable** (i.e., it directly maps to a single table and does not contain complex operations like JOIN, GROUP BY, or DISTINCT), you can insert data into the underlying table via the view.
* **Non-Updatable Views:** If the view is **non-updatable** (e.g., it involves complex queries), you cannot insert rows directly into the view.

**Example of an Updatable View:**

CREATE VIEW ActiveEmployees AS

SELECT EmployeeID, Name, Salary FROM Employees WHERE Status = 'Active';

You can insert a new row into the Employees table via the ActiveEmployees view if the view is simple and only references one table.

INSERT INTO ActiveEmployees (EmployeeID, Name, Salary)

VALUES (101, 'John Doe', 50000);

**2. Deleting from a View:**

* **Updatable Views:** If the view is updatable (i.e., it directly references a single table without complex operations), you can delete rows from the underlying table through the view.
* **Non-Updatable Views:** If the view is non-updatable, you cannot delete rows through the view.

**Example of Deleting from an Updatable View:**

DELETE FROM ActiveEmployees WHERE EmployeeID = 101;

**Summary:**

* **Yes**, you can **insert** or **delete** rows in a view if the view is updatable (i.e., it references a single table and doesn't use complex clauses).
* **No**, you cannot insert or delete rows in a view if it is non-updatable (e.g., the view contains JOIN, GROUP BY, DISTINCT, etc.).

**--------------------------------------------------------**

CREATE VIEW ActiveEmployees AS

SELECT EmployeeID, Name, Salary

FROM Employees

WHERE Status = 'Active'

WITH CHECK OPTION;

the **WITH CHECK OPTION** ensures that any attempt to insert or update rows through the ActiveEmployees view must satisfy the condition that the employee's Status is 'Active'.

The purpose of the WITH CHECK option is to ensure that all UPDATE and INSERTs satisfy the conditions in the view definition.

When working with views, any column from the base table that has a **NOT NULL** constraint must be included in the view to allow an **INSERT** operation to work.

DELETE FROM Employee\_View WHERE age > 60;

DROP VIEW Employee\_View;

There are 2 types of Views in SQL: Simple View and Complex View. Simple views constitute only of a single base table. Complex views can be constituted from more than one base table.

A **complex view** in SQL refers to a view that is built using complex queries, which might include multiple tables, joins, aggregations, subqueries, and/or advanced filtering conditions. These views are often used to represent a complex set of data in a simplified and reusable way, allowing users to query the view without needing to write complex SQL statements every time.

Here are some examples of **complex views**:

**1. View with Joins**

A view that combines data from multiple tables using **JOINs** (INNER JOIN, LEFT JOIN, etc.).

Example:

CREATE VIEW Employee\_Salary\_Info AS

SELECT e.Employee\_ID, e.Name, e.Department, s.Salary

FROM Employees e

INNER JOIN Salaries s ON e.Employee\_ID = s.Employee\_ID;

This view combines data from the Employees and Salaries tables, showing each employee's ID, name, department, and salary.

**2. View with Aggregations**

A view that calculates aggregate values, such as sums, averages, counts, etc.

Example:

CREATE VIEW Department\_Salary\_Summary AS

SELECT Department, AVG(Salary) AS Average\_Salary, COUNT(Employee\_ID) AS Employee\_Count

FROM Employees

GROUP BY Department;

This view summarizes average salaries and employee counts for each department.

**3. View with Subqueries**

A view that uses a subquery to filter or calculate data.

Example:

CREATE VIEW Top\_Earning\_Employees AS

SELECT e.Employee\_ID, e.Name, e.Salary

FROM Employees e

WHERE e.Salary > (SELECT AVG(Salary) FROM Employees);

This view shows employees with salaries above the average salary in the company.

**4. View with Multiple Joins and Aggregations**

A more complex view combining multiple joins and aggregate functions.

Example:

CREATE VIEW Sales\_Overview AS

SELECT p.Product\_Name, SUM(o.Quantity) AS Total\_Sold, SUM(o.Quantity \* o.Unit\_Price) AS Total\_Revenue

FROM Products p

INNER JOIN Orders o ON p.Product\_ID = o.Product\_ID

GROUP BY p.Product\_Name;

This view aggregates sales data by product, showing the total quantity sold and total revenue per product.

**5. View with Case Statements**

A view that uses **CASE** statements to categorize or manipulate data based on certain conditions.

Example:

CREATE VIEW Employee\_Performance AS

SELECT Employee\_ID, Name,

CASE

WHEN Performance\_Score >= 90 THEN 'Excellent'

WHEN Performance\_Score >= 75 THEN 'Good'

WHEN Performance\_Score >= 50 THEN 'Average'

ELSE 'Needs Improvement'

END AS Performance\_Rating

FROM Employees;

This view assigns a performance rating to employees based on their performance score.

**Advantages of Using Complex Views:**

1. **Simplifies Complex Queries:** Instead of writing long and complex queries repeatedly, you can define a view once and refer to it later, saving time and effort.
2. **Data Abstraction:** Views hide the complexity of the underlying data structure, presenting users with a simpler interface.
3. **Reusability:** Once a complex view is created, it can be reused by multiple queries or applications without redefining the logic each time.
4. **Security:** Views can provide a restricted view of the data, allowing users to see only the specific data they need without exposing the entire underlying tables.

**Disadvantages:**

1. **Performance:** Complex views can sometimes lead to performance issues if the underlying queries are inefficient or the view involves large tables and complex operations.
2. **Maintenance:** If the structure of the underlying tables changes, views may need to be modified to adapt to those changes.

In summary, complex views are powerful tools for abstracting complex database logic, making querying easier for end users, and simplifying the development process. However, they should be used carefully to avoid performance bottlenecks.

In SQL, **views** are virtual tables that represent the result of a stored query. They allow users to simplify complex queries, restrict access to sensitive data, and provide a convenient way to reuse common queries. Views are widely used in real-world applications for several practical reasons:

**1. Simplifying Complex Queries**

* **Real-Time Application**: Views are used to simplify complex queries that involve multiple joins, aggregations, or subqueries. Instead of writing the same complex query multiple times, you can define it as a view and reuse it whenever needed.
* **Example**: Suppose you frequently need to retrieve customer details along with their recent orders. Instead of writing the same JOIN and WHERE logic in every query, you can create a view to encapsulate that logic.

CREATE VIEW customer\_orders AS

SELECT c.customer\_id, c.customer\_name, o.order\_id, o.order\_date

FROM customers c

JOIN orders o ON c.customer\_id = o.customer\_id

WHERE o.order\_date > '2023-01-01';

Now, you can use the customer\_orders view in queries:

SELECT \* FROM customer\_orders WHERE customer\_id = 101;

**2. Data Security and Access Control**

* **Real-Time Application**: Views are often used to **restrict access to sensitive data** by providing a limited subset of columns or rows. You can create views that expose only the necessary data to users, ensuring that sensitive information is hidden.
* **Example**: For a company with employees' sensitive salary information, you can create a view that excludes salary details when providing access to certain users.

CREATE VIEW employee\_public\_info AS

SELECT employee\_id, name, department

FROM employees;

Now, users can access the employee\_public\_info view without seeing sensitive data like salaries.

**Database User Permissions:**

REVOKE SELECT ON employees.salary FROM public\_user;

GRANT SELECT (employee\_id, name, department) ON employees TO public\_user;

**Row-Level Security (RLS):**

CREATE POLICY public\_employee\_policy

ON employees

FOR SELECT

USING (NOT sensitive\_column);

**3. Providing Abstraction**

* **Real-Time Application**: Views can provide an abstraction layer, hiding the complexity of database schema changes from end-users. When the underlying database structure changes (e.g., if a table is split or a new column is added), you can modify the view without requiring users to update their queries.
* **Example**: If the company's database structure changes and new columns or tables are added, you can update the view without requiring each user to modify their individual queries.

CREATE OR REPLACE VIEW customer\_details AS

SELECT customer\_id, customer\_name, email, phone

FROM customers;

This way, users can continue using customer\_details without worrying about changes in the underlying schema.

**4. Data Aggregation and Reporting**

* **Real-Time Application**: Views are commonly used in **reporting and data aggregation** scenarios. A view can encapsulate aggregated data (e.g., total sales, average order value) to provide summary reports that users can query directly.
* **Example**: A business might create a view to show monthly sales totals for reporting purposes.

CREATE VIEW monthly\_sales AS

SELECT MONTH(order\_date) AS month, SUM(order\_total) AS total\_sales

FROM orders

GROUP BY MONTH(order\_date);

Users can then query this view for monthly sales figures:

SELECT \* FROM monthly\_sales WHERE month = 5;

**5. Reusing Common Logic**

* **Real-Time Application**: If multiple applications or users need the same data but with a consistent format, views can be used to define the **common logic** for fetching that data. This reduces the chances of errors and ensures data consistency across applications.
* **Example**: A company’s HR system might use a view to combine employee details and performance scores from different tables, providing a unified interface for HR staff.

CREATE VIEW employee\_performance AS

SELECT e.employee\_id, e.name, p.performance\_score

FROM employees e

JOIN performance\_reviews p ON e.employee\_id = p.employee\_id;

**6. Handling Historical Data and Temporal Queries**

* **Real-Time Application**: Views can be used to manage and present **historical data** or **temporal queries**. For instance, when dealing with historical records (e.g., sales data over time), views can provide a snapshot of data at a certain point in time or calculate moving averages.
* **Example**: A company may use a view to track the historical prices of products and provide the most recent price in reports.

CREATE VIEW product\_price\_history AS

SELECT product\_id, price, change\_date

FROM product\_price\_changes

WHERE change\_date <= CURRENT\_DATE;

**7. Data Integration Across Different Sources**

* **Real-Time Application**: In some cases, data might reside in different tables or even in different databases. Views can be used to **integrate data** from different sources, presenting a unified result set.
* **Example**: In a system with separate customer and order databases, a view can combine both datasets to make it easier for analysts to generate reports.

CREATE VIEW customer\_order\_details AS

SELECT c.customer\_id, c.customer\_name, o.order\_id, o.order\_date

FROM external\_customers c

JOIN external\_orders o ON c.customer\_id = o.customer\_id;

**Conclusion:**

In real-world applications, views serve a wide range of purposes, including simplifying complex queries, improving security by limiting data access, providing abstraction from database schema changes, and supporting aggregation and reporting. Views are an essential tool in managing data access and optimizing query performance, especially in large and complex database systems.

**indexes** are used for optimizing performance, particularly for searching and sorting, while **views** are used for data abstraction, security, and simplifying complex queries.

**Why a View Is Preferred:**

* **Abstraction**: Views encapsulate the logic of querying sensitive data, ensuring users cannot access the sensitive columns even if they have access to the underlying tables.
* **Reusability**: Once the view is created, it can be reused in multiple queries, and the logic does not need to be rewritten.
* **Simplified Access Control**: Views can provide an easy-to-manage layer for controlling access to non-sensitive data, especially when the underlying schema changes.

While the alternatives can work, **views** provide a simpler and more flexible way to manage such use cases by abstracting the logic into a virtual table that enforces security and reduces the chances of accidental data exposure.

the view dynamically fetches the latest data from the table each time you query it.