**TASK 7**

1. **What is View ?**

A **View** is a **virtual table** based on the result of a **SQL query**.

* It does **not store data** itself.
* It presents data from one or more tables.
* Used to **simplify complex queries**, **enhance security**, and **provide abstraction**.

**Example in SQL:**

CREATE VIEW EmployeeView AS

SELECT name, department

FROM Employees

WHERE status = 'Active';

This view shows only active employees and can be queried like a regular table:

**SELECT \* FROM EmployeeView;**

1. **Can we update data through view ?**

Yes, **you can update data through a view** in SQL, but with some **important limitations**.

**✅ You can update data through a view if:**

* The view is **simple** — based on a single table.
* The view **does not** use:
  + JOIN
  + GROUP BY
  + DISTINCT
  + Aggregate functions (e.g., SUM, COUNT)
  + UNION
* The columns being updated are **directly mapped** to columns in the base table.

Example:

CREATE VIEW vw\_employee AS

SELECT id, name, salary

FROM employees;

You can do:

UPDATE vw\_employee

SET salary = 60000

WHERE id = 101;

1. **What is a materialized view?**

A **materialized view** is a **database object** that stores the **result of a query physically on disk**. Unlike a regular view (which is just a saved query and runs every time it's accessed), a materialized view **actually stores the data** and can be refreshed periodically.

### Example

CREATE MATERIALIZED VIEW sales\_summary AS

SELECT region, SUM(sales) AS total\_sales

FROM sales

GROUP BY region;

To update the data:

REFRESH MATERIALIZED VIEW sales\_summary;

1. **Difference between view and table?**

| **Feature** | **Table** | **View** |
| --- | --- | --- |
| **Definition** | A table is a **physical object** that stores **data** in rows and columns. | A view is a **virtual table** based on a **query**; it doesn't store data by default. |
| **Storage** | Stores data **physically on disk**. | Does **not store data** (unless it's a materialized view). |
| **Data** | Contains actual, independent data. | Shows data from one or more tables via a SELECT query. |
| **Updates** | Fully updatable. | Updatable **only if** it's a simple view. Complex views are read-only. |
| **Performance** | Fast — direct access to stored data. | Slower — query runs each time (unless materialized). |
| **Use Case** | For storing and managing actual data. | For **simplifying queries**, **hiding complexity**, or **adding security**. |
| **Dependency** | Independent. | Dependent on underlying tables. |
| **Security** | Permissions needed to access raw data. | Can restrict access to certain columns/rows by exposing only part of the data. |

1. **How to drop a view?**

To delete (drop) a view from the database, use the **DROP VIEW** statement.

### ✅ ****Syntax (Standard SQL):****

DROP VIEW view\_name;

### ✅ ****Examples:****

#### 1. Drop a single view:

DROP VIEW high\_salary\_employees;

#### 2. Drop multiple views at once (if supported by your DBMS):

DROP VIEW view1, view2;

### 💡 Notes:

* **The view must exist**, or you'll get an error (unless you use IF EXISTS).
* **Permissions**: You need appropriate privileges to drop a view.

### ✅ Using IF EXISTS (to avoid error if view doesn't exist):

#### PostgreSQL / MySQL / SQL Server (from 2016+):

DROP VIEW IF EXISTS high\_salary\_employees;

1. **Why use views?**

Views are **virtual tables** created from queries on one or more real tables. They offer many practical benefits in database design, development, and security.

### 🔍 ****Top Reasons to Use Views****

#### 1. ✅ **Simplify Complex Queries**

* Views can wrap **complex joins, filters, and calculations** into a single, easy-to-use object.
* Helps avoid repeating the same logic in multiple queries.

**Example:**

CREATE VIEW active\_customers AS

SELECT id, name

FROM customers

WHERE status = 'active';

#### 2. 🔐 **Enhance Data Security**

* Views can **restrict access** to sensitive columns or rows by exposing only specific data.

**Example:**  
Instead of giving access to the full employees table, you can create:

CREATE VIEW employee\_public AS

SELECT name, department

FROM employees;

#### 3. 🧼 **Maintain Data Abstraction**

* Allows you to **hide the complexity** of the database structure from end users.
* You can change the underlying table structure without affecting users of the view (as long as the view definition remains valid).

#### 4. 📈 **Reuse Query Logic**

* Write once, use many times.
* Developers and analysts can reuse views instead of duplicating complex queries.

#### 5. 🛡️ **Data Consistency**

* Views provide a **centralized, consistent way** to access data based on predefined logic.
* Reduces the risk of errors or inconsistencies in different parts of the application.

#### 6. 🔄 **Support Logical Data Independence**

* Applications using views don’t need to change even if the underlying table changes (columns renamed, added, or dropped), as long as the view stays valid.

1. **Can we create indexed views?**

### ****Yes, you can create indexed views**** — but only in ****some databases****, and with ****certain rules and restrictions****.

### 🔍 What is an Indexed View?

An **indexed view** is a view that has a **unique clustered index** created on it, which means:

* The **view stores data physically** (like a table).
* It behaves like a **materialized view** — much faster to query than a regular view.
* The index helps improve performance, especially on complex joins or aggregations.

## 🧠 Benefits of Indexed Views

| **Benefit** | **Description** |
| --- | --- |
| 🔄 Stored on Disk | View results are materialized and stored |
| ⚡ Faster Performance | Great for reporting and analytics |
| 🔍 Can Be Auto-Used | Optimizer may use indexed views in queries |

## 💻 Supported Databases & Syntax

### ✅ ****SQL Server****

Yes, supports **indexed views** (called "materialized views" in other systems).

#### Example:

CREATE VIEW sales\_summary

WITH SCHEMABINDING AS

SELECT region, COUNT\_BIG(\*) AS total\_sales

FROM dbo.sales

GROUP BY region;

Now create a unique clustered index on the view:

CREATE UNIQUE CLUSTERED INDEX idx\_sales\_summary

ON sales\_summary(region);

1. **How to secure data using views?**

### 1. 🔎 ****Expose Only Specific Columns****

Limit users to seeing **only certain columns** — hide sensitive fields like passwords, salaries, SSNs, etc.

#### Example:

-- Original table has sensitive data

SELECT \* FROM employees;

-- Secure view hides salary and SSN

CREATE VIEW employee\_public AS

SELECT id, name, department

FROM employees;

✅ Users granted access to employee\_public can’t see salary or SSN.

### 2. 🧾 ****Filter Rows with WHERE Clause****

Limit access to **only certain rows** using filters (e.g. department, region, user ID).

#### Example:

CREATE VIEW hr\_employees AS

SELECT id, name, department

FROM employees

WHERE department = 'HR';

✅ Only HR data is visible through the view.

### 3. 🧍‍♂️ ****User-Specific Access (with CURRENT\_USER or SESSION\_USER)****

In some databases, you can create views that filter data based on the **current user**.

#### PostgreSQL Example:

CREATE VIEW my\_data AS

SELECT \*

FROM records

WHERE owner = CURRENT\_USER;

Each user sees **only their own data**.

### 4. 👮 ****Combine Views with GRANT / REVOKE****

Use **views + permissions** to restrict access:

Allow user to read from the view

GRANT SELECT ON employee\_public TO analyst;

Deny access to the base table

REVOKE SELECT ON employees FROM analyst;

✅ User can access limited data in employee\_public, but not the full employees table.

### 5. 🔄 ****Prevent Updates Using Read-Only Views****

By default, some views are **read-only**, especially if they include:

* Joins
* Aggregations
* GROUP BY

This can help prevent unauthorized data modification

1. **What are limitations of views?**

While **views** are powerful tools for abstraction, security, and simplifying complex queries, they **have several limitations** you should be aware of — especially when trying to update data or optimize performance.

## 🔒 **1. Read-Only in Many Cases**

* Views are often **not updatable** if they include:
  + JOIN
  + GROUP BY
  + DISTINCT
  + Aggregate functions (SUM, AVG, etc.)
  + UNION / UNION ALL
  + Subqueries in the SELECT list

✅ **Updatable only** if the view is based on a **single table** and doesn't include the features above.

## 🚫 **2. Cannot Store Data (Unless Materialized)**

* Standard views are **virtual** and do **not store data**.
* Every time you query a view, it re-runs the underlying query.
* This can cause **performance issues** if the query is complex.

💡 Use **materialized views** or **indexed views** if you need stored data.

## ⚙️ **3. No Indexes on Regular Views**

* You **cannot create indexes** on standard views.
* Indexed/materialized views are supported in some databases, but with strict rules.

## 🧱 **4. Dependency on Base Tables**

* If the **structure of underlying tables changes** (e.g., column renamed or dropped), the view can become **invalid**.
* Changes to base data **immediately affect** view results.

## 🔐 **5. Limited Support for Triggers**

* You **cannot create triggers** on views in many databases (like MySQL, PostgreSQL).
* Some databases (like SQL Server, Oracle) support **INSTEAD OF triggers** to make views updatable.

## 🔁 **6. Cannot Pass Parameters**

* Views **cannot accept parameters** like stored procedures or functions can.
* If you want dynamic filtering, you must apply WHERE clauses in your query, not inside the view.

## 📉 **7. Performance Overhead**

* If the view is built on top of **complex joins or aggregations**, querying it can be **slow** — especially if the view is used inside larger queries

1. **How does WITH CHECK OPTION work?**

The WITH CHECK OPTION is a **constraint** you can add when creating a **view** to ensure that **any data inserted or updated through the view still matches the view’s condition** (i.e., the WHERE clause).

## ✅ **What It Does**

It **prevents users** from:

* Inserting rows **that don’t satisfy** the view’s WHERE condition.
* Updating existing rows **in a way that would remove them** from the view.

### 🧠 ****Why Use It?****

Without WITH CHECK OPTION, someone could:

* Insert or update data **through the view** that **won’t be visible** in the view.
* This could lead to confusion or unintended data inconsistencies.

## 🧪 **Example:**

### Step 1: Base table

CREATE TABLE employees (

id INT,

name VARCHAR(100),

department VARCHAR(50)

);

### Step 2: Create view (only HR employees)

CREATE VIEW hr\_employees AS

SELECT \* FROM employees

WHERE department = 'HR'

WITH CHECK OPTION;

### ✅ Allowed:

INSERT INTO hr\_employees (id, name, department)

VALUES (1, 'Alice', 'HR'); -- OK ✅

### ❌ Not Allowed:

INSERT INTO hr\_employees (id, name, department)

VALUES (2, 'Bob', 'Finance'); -- ❌ ERROR: fails check option

### ❌ Also Not Allowed:

UPDATE hr\_employees

SET department = 'IT'

WHERE id = 1; -- ❌ ERROR: row would no longer be visible in the view

## 🔁 Types of CHECK OPTION (in some DBMSs like Oracle):

1. **WITH CHECK OPTION** – applies only to **this** view
2. **WITH CASCADED CHECK OPTION** – applies to **this view and all underlying views**
3. **WITH LOCAL CHECK OPTION** – applies **only to the current view**, not underlying views