SQL Views

SQL views are virtual tables that are created using a SELECT statement in SQL. A view is a database object that acts as a filter to the data stored in one or more tables. It is a logical representation of data in a database that can be used to simplify the complexity of data and enhance security.

Syntax

Create view   v_name

As

Select * column 1 and column 2

From table_name

Where condition;

Advantages to using views:

-- views can simplify complex queries and make them easier to read and understand.

-- Views can also be used to enhance security by restricting access to certain data.

-- views can be used to hide sensitive data from certain users or applications.

Types of Views in SQL Server:

1- *Simple View is a view created from a single table and does not use complex SQL features.*

- *Is based on only one table*
- ***Does not use**:*
- *JOIN*
- *GROUP BY*
- *HAVING*
- *Aggregate functions (SUM, AVG, COUNT, etc.)*

Syntax

CREATE VIEW name_View AS

SELECT column1, column2

FROM table_name

WHERE condition;

Real-life use cases

1️⃣ User Access & Security (Most Common Use)

Problem:
You don't want users to see sensitive columns (salary, password, SSN).

Solution (Simple View):

```
CREATE VIEW Employee_Public AS
SELECT EmployeeID, Name, Department
FROM Employee;
```

2️⃣ Active Records Only (Filtering Data)

Problem:
Applications frequently need **only active records**.

Solution:

```
CREATE VIEW ActiveCustomers AS
SELECT CustomerID, CustomerName, Email
FROM Customers
WHERE Status = 'Active';
```

2- Indexed View is a view that has a clustered index created on it, which means:

the result of the view is stored physically on disk, just like a table.

Indexed Views are used to:

- Improve performance
- Speed up JOINs and aggregate queries
- Reduce repeated calculations
- Optimize reporting queries

Key Difference from Normal Views

Feature	Normal View	Indexed View
Stores data	❌ No (virtual)	✅ Yes (physical)

Performance	Normal	🚀 Faster
Uses index	❌ No	✅ Yes
SQL Server support	✅	✅ (only SQL Server)

== When to Use Indexed Views?

✓ Large tables
✓ Heavy aggregation (`SUM`, `COUNT`, `AVG`)
✓ Frequently executed queries
✓ Reporting systems

==When NOT to Use Them?

❌ Tables with frequent INSERT/UPDATE/DELETE
❌ Small datasets
❌ When schema changes often

Real-life use cases

1️⃣ E-Commerce Website (Fast Sales Dashboards)

Problem:
Managers frequently check **total sales per product/category**.
Running `SUM()` and `GROUP BY` on millions of rows is slow.

Indexed View Solution:

✓ Dashboard loads instantly
✓ Aggregates are already stored
✓ Very common in large e-commerce systems

2️⃣ Banking System (Account Balance Summaries)

Problem:
Balance is calculated from thousands of transactions every time.

Indexed View Solution:

- View stores **total deposits and withdrawals per account**
- Clustered index keeps data ready

✓ Faster balance checks
✓ Used in ATMs and online banking
✓ Avoids recalculating totals repeatedly

3--University / Education System (Results & GPA)

Problem:
Calculating GPA for thousands of students repeatedly is expensive.

Indexed View Solution:

- Pre-calculated **GPA per student**
- Indexed for fast access

✓ Faster student portals
✓ Efficient result processing
✓ Stable reporting

3- Partitioned View a view that combines data from multiple tables that have the same structure, usually to make them behave like one large table.

Partitioned Views are used to:

- Manage very large tables
- Improve performance
- Separate data logically (by year, region, department, etc.)
- Support scalability

How does it work?

- Data is split across multiple tables
- Each table contains different rows
- A CHECK constraint defines which rows belong to each table
- A view combines them using `UNION ALL`

Example Scenario:

```
CREATE TABLE Orders_2023 (

   OrderID INT PRIMARY KEY,

   OrderDate DATE,

   Amount DECIMAL(10,2),

   CHECK (YEAR(OrderDate) = 2023)

);

CREATE TABLE Orders_2024 (

   OrderID INT PRIMARY KEY,

   OrderDate DATE,

   Amount DECIMAL(10,2),

   CHECK (YEAR(OrderDate) = 2024)

);
```

Create the Partitioned View

```
CREATE VIEW Orders_All

AS

SELECT * FROM Orders_2023

UNION ALL

SELECT * FROM Orders_2024;
```

Partitioned View vs Indexed View

Feature	Partitioned View	Indexed View
Purpose	Split large data	Speed up queries
Data storage	Separate tables	Stored via index
Uses UNION ALL	✅ Yes	❌ No
Performance gain	Scalability	Query speed

Real-life use cases

1️⃣ Banking System (Transactions by Year)

Problem:
A bank has **millions of transactions**. One table becomes very slow.

Solution using Partitioned View:

- `Transactions_2023`
- `Transactions_2024`
- `Transactions_2025`

Each table stores one year of data.

```
CREATE VIEW AllTransactions AS
SELECT * FROM Transactions_2023
UNION ALL
SELECT * FROM Transactions_2024
UNION ALL
SELECT * FROM Transactions_2025;
```

2️⃣ University / College Database (Students by Batch)

Problem:
Student records grow every year.

Solution:

- `Students_2022`
- `Students_2023`
- `Students_2024`

```
CREATE VIEW AllStudents AS
SELECT * FROM Students_2022
UNION ALL
SELECT * FROM Students_2023
UNION ALL
SELECT * FROM Students_2024;
```

3️⃣ E-Commerce System (Orders by Region)

Problem:
Orders are stored globally → slow queries.

Solution:

- `Orders_US`
- `Orders_EU`
- `Orders_ASIA`

```
CREATE VIEW AllOrders AS
SELECT * FROM Orders_US
UNION ALL
SELECT * FROM Orders_EU
UNION ALL
SELECT * FROM Orders_ASIA;
```

. Can We Use DML (INSERT, UPDATE, DELETE) on Views? Yes we can

Which types of views allow DML operations?

=simple type

= View with check option

= Join View

What are the restrictions or limitations when performing DML on a view?

1. **View must be updatable**
   a. Only **simple views** (based on one table, no aggregates, no `DISTINCT`, no `GROUP BY`, no `UNION`) are fully updatable.
   b. Views that are **complex** (joins, aggregates, distinct, union, computed columns) are either partially updatable or non-updatable.
2. **Columns must belong to a single base table for updates**
   a. If the view is a join, you can update **only the columns from one underlying table** at a time.

3. **Cannot modify derived/computed columns**

```
CREATE VIEW vw_Salary
AS
SELECT EmployeeID, Salary * 1.1 AS NewSalary
FROM Employees;
```

   a. ❌ Cannot insert or update `NewSalary` because it's computed.

4. **Cannot perform DML that violates view constraints**
   a. If the view has a `WITH CHECK OPTION`, you cannot insert/update rows that **would be excluded from the view**.

Give at least one real-life example where updating a view is useful (e.g., HR system, e-commerce orders, etc.)

Scenario: HR System – Updating Employee Contact Info

Problem

- HR has a large **Employees** table with sensitive columns (Salary, Social Security Number, etc.).
- HR staff need to **update only contact information** (phone, email) without seeing or touching sensitive data.

Step 1: Create a Simple View for Safe Updates

```
CREATE VIEW vw_EmployeeContacts
AS
SELECT EmployeeID, Name, Phone, Email
FROM Employees;
```

Step 2: Update via the View

```
UPDATE vw_EmployeeContacts
SET Phone = '987-654-3210',
    Email = 'new.email@example.com'
WHERE EmployeeID = 101;
```

2. How Can Views Simplify Complex Queries?

- Explain how a View can help simplify JOIN-heavy queries.

1️⃣ Problem: JOIN-heavy Queries Are Complex

When a query involves **many tables**, joins, and conditions, it can become long, hard to read, and error-prone.

- Multiple joins make the query **long and difficult to maintain**.
- Rewriting it for reports or analytics can be tedious.

2️⃣ Solution: Use a View to Encapsulate Joins

- A view can **predefine the joins**, so users don't have to rewrite them every time.
- Makes queries **shorter, readable, and maintainable**.

Create an example view that joins at least two of your banking tables, such as: o Customer + Account o Account + Transaction

1️⃣ View: Customer + Account

```
CREATE VIEW vw_CustomerAccounts

AS

SELECT

   c.CustomerID, c.FullName, c.Email,   a.AccountID, a.AccountType,  a.Balance,  a.OpenDate

FROM Customers c

JOIN Accounts a

   ON c.CustomerID = a.CustomerID;
```

2️⃣ View: Account + Transaction

```
CREATE VIEW vw_AccountTransactions

AS

SELECT

 a.AccountID,  a.AccountType, a.Balance AS CurrentBalance, t.TransactionID,
   t.TransactionDate, t.Amount,   t.TransactionType

FROM Accounts a

JOIN Transactions t

   ON a.AccountID = t.AccountID;
```