Part 1: Research & Documentation

Types of Views in SQL Server

1. Standard View (Regular View)

What is it?

A **Standard View** is a virtual table created using a SELECT statement.
It does **not store data physically**—it only stores the query definition and retrieves data from base tables when queried.

Key Differences

- Data is **not stored** in the view
- Always reflects **current data** from underlying tables
- Simplest and most commonly used view type
- No performance improvement by itself
- **Real-Life Use Case**

University System

- A view that shows student name, course name, and grade by joining multiple tables.
- Used to simplify complex queries for lecturers or admins.

Example:

```
CREATE VIEW vw_StudentResults AS
SELECT s.StudentName, c.CourseName, r.Grade
FROM Students s
JOIN Results r ON s.StudentID = r.StudentID
JOIN Courses c ON r.CourseID = c.CourseID;
```

Limitations & Performance Considerations

- No data storage → query runs every time
- Performance depends on underlying tables and joins

- Cannot use indexes directly on the view
- Heavy queries may be slow on large datasets

2. Indexed View (Materialized View)

What is it?

An **Indexed View** stores the result set **physically on disk** by creating a **clustered index** on the view.
SQL Server maintains the view data automatically.

Key Differences

- Data **is stored physically**
- Improves performance for **complex aggregations**
- Requires `SCHEMABINDING`
- Automatically updated when base tables change

Real-Life Use Case

Banking System

- A view that calculates **total balance per branch**
- Used frequently in reports and dashboards

Example:

```
CREATE VIEW vw_BranchBalance
WITH SCHEMABINDING
AS
SELECT BranchID, SUM(Balance) AS TotalBalance
FROM dbo.Accounts
GROUP BY BranchID;

CREATE UNIQUE CLUSTERED INDEX idx_BranchBalance
ON vw_BranchBalance (BranchID);
```

Limitations & Performance Considerations

- Slower INSERT, UPDATE, DELETE operations
- Strict rules (no `SELECT *`, no outer joins, no non-deterministic functions)
- Requires additional storage
- Best for **read-heavy systems**, not write-heavy ones

3. Partitioned View (Union View)

What is it?

A **Partitioned View** combines data from **multiple tables** (usually partitioned by range) using `UNION ALL`.
Each table stores a portion of the data.

Key Differences

- Data stored in **multiple tables**
- Uses `UNION ALL`
- Useful for **horizontal partitioning**
- Improves performance by scanning only relevant tables

Real-Life Use Case

E-Commerce System

- Orders split by year into separate tables:
  - Orders_2023
  - Orders_2024
- The view allows querying all orders as one table

Example:

```
CREATE VIEW vw_AllOrders AS
SELECT * FROM Orders_2023
UNION ALL
SELECT * FROM Orders_2024;
```

Limitations & Performance Considerations

- Tables must have identical structure
- No automatic partition management
- Complex maintenance as data grows
- Poor design can hurt performance

Which Types of Views Allow DML Operations?

☑ Standard (Simple) Views

✓ **Allow DML** *if certain conditions are met*

✓ Most commonly used for INSERT, UPDATE, DELETE

Updatable when:

- View is based on **one base table**
- No aggregation or grouping
- No joins
- No DISTINCT
- No computed columns
- No GROUP BY / HAVING

📌 SQL Server directly updates the underlying table.

❌ Indexed Views

❌ **DML not allowed directly on the view**

- You **cannot INSERT/UPDATE/DELETE** data through an indexed view
- You must modify the **base tables**
- SQL Server automatically maintains the indexed view

❌ Partitioned Views (UNION ALL Views)

⚠ **Limited DML support**

✓ INSERT allowed **only if**:

- The inserted row belongs to **one specific partition**
- A CHECK constraint exists to identify the correct table

❌ UPDATE/DELETE often restricted and complex

2️⃣ Restrictions and Limitations for DML on Views

DML operations **are NOT allowed** on a view if it contains:

🚫 JOINs (multiple tables)
🚫 GROUP BY or HAVING
🚫 Aggregate functions (SUM, COUNT, AVG…)
🚫 DISTINCT
🚫 UNION / UNION ALL (except limited INSERT)
🚫 Computed or derived columns
🚫 Subqueries in SELECT
🚫 TOP with ORDER BY

📌 **Key Rule:**

If SQL Server cannot clearly map a change to **one row in one base table**, DML is blocked.

3️⃣ Real-Life Examples of Updating Views

👩‍💼 HR System Example

Table: Employees

| EmployeeID | Name | Salary | Department | IsActive |

View: ActiveEmployeesView

```
CREATE VIEW ActiveEmployeesView AS
SELECT EmployeeID, Name, Salary
FROM Employees
WHERE IsActive = 1;
```

✓ HR updates salaries **through the view**:

```
UPDATE ActiveEmployeesView
SET Salary = Salary + 200
WHERE EmployeeID = 101;
```

📌 The update affects the **Employees** table directly.

✅ Benefits:

- Security (HR can't see inactive employees)
- Simpler interface
- Controlled access

🛒 E-Commerce Orders Example

View: PendingOrdersView

```
CREATE VIEW PendingOrdersView AS
SELECT OrderID, CustomerID, Status
```

```
FROM Orders
WHERE Status = 'Pending';
```

✓ Mark an order as shipped:

```
UPDATE PendingOrdersView
SET Status = 'Shipped'
WHERE OrderID = 5001;
```

2. How Can Views Simplify Complex Queries?

How Views Simplify JOIN-Heavy Queries

A **view** is a stored SELECT statement that behaves like a virtual table.

Key benefits for complex JOINs:

- ✅ **Removes repeated JOIN logic**
  Write the JOIN once → reuse it everywhere.
- ✅ **Improves readability**
  Call center agents or analysts query a simple view instead of a long SQL statement.
- ✅ **Reduces errors**
  Less copy-paste = fewer JOIN mistakes.
- ✅ **Security control**
  Users can access the view without touching base tables.

- ✅ **Centralized changes**
  If table structure changes, update the view—not every query.

2️⃣ Banking Scenario: Call Center Account Summary

Business need

Call center agents frequently need to see:

- Customer name
- Account number
- Account type
- Balance
- Account status

Instead of writing JOINs every time, we create a **view**.

3️⃣ Example Tables (Simplified)

```
Customers (
    CustomerID INT,
    FullName VARCHAR(100),
    Phone VARCHAR(20)
);

Accounts (
    AccountID INT,
    CustomerID INT,
    AccountType VARCHAR(20),
    Balance DECIMAL(10,2),
    Status VARCHAR(20)
);
```

4️⃣ JOIN-Heavy Query (Without a View)

```
SELECT
    c.CustomerID,
    c.FullName,
    a.AccountID,
    a.AccountType,
    a.Balance,
    a.Status
FROM Customers c
JOIN Accounts a
    ON c.CustomerID = a.CustomerID;
```

❌ Problem:
This query must be rewritten **every time** an agent or report needs account details.

5️⃣ Creating a View to Simplify the Query

```
CREATE VIEW vw_CustomerAccountSummary
AS
SELECT
    c.CustomerID,
    c.FullName,
    c.Phone,
    a.AccountID,
    a.AccountType,
    a.Balance,
    a.Status
FROM Customers c
JOIN Accounts a
    ON c.CustomerID = a.CustomerID;
```

✓ JOIN logic is now stored once.

6️⃣ Using the View (Much Simpler!)

```
SELECT *
FROM vw_CustomerAccountSummary
WHERE Status = 'Active';
```

Or:

```
SELECT FullName, AccountType, Balance
FROM vw_CustomerAccountSummary
WHERE CustomerID = 101;
```

✅ No JOINs
✅ Easy to read
✅ Faster to write
✅ Perfect for call center dashboards

7️⃣ Real-Life Benefit for Call Center Agents

Without View	With View
Long JOIN queries	Simple SELECT
Higher risk of errors	Consistent results
Hard to maintain	Easy to update
Requires SQL expertise	Beginner-friendly

8️⃣ Summary

- Views **simplify complex JOIN-heavy queries**
- They act as **pre-built query templates**

- Ideal for **frequently accessed data** like banking account summaries
- Improve **readability, security, and maintainability**