SQL Views – Research & Implementation Task

Part 1: Types of Views in SQL Server

A **view** in SQL Server is a virtual table created from a SELECT query. Views are used to simplify queries, enhance security, and improve data organization.

1. Standard View (Regular View)

What is it?

A **Standard View** is a virtual table based on a SELECT statement.
It does **not store data physically**; it retrieves data from underlying tables each time it is queried.

```
CREATE VIEW vw_Students
AS
SELECT StudentID, Name, Department
FROM Students;
```

Key Differences

- Data is **not stored** in the view
- Most flexible and commonly used view type
- Performance depends entirely on base tables

Real-Life Use Cases

- **University system:** Display student names and departments without exposing grades
- **Banking:** Show account numbers and balances without revealing customer details
- **HR system:** Show employee information while hiding salary

Limitations & Performance Considerations

- No performance gain for complex queries
- Cannot use ORDER BY (unless combined with TOP)
- Limited support for INSERT, UPDATE, and DELETE
- Performance depends on indexes of base tables

2. Indexed View (Materialized View)

What is it?

An **Indexed View** stores the query result **physically on disk** by creating a **unique clustered index** on the view.
This allows faster query execution for complex aggregations.

```
CREATE VIEW vw_TotalRevenue
WITH SCHEMABINDING
AS
SELECT ProductID, SUM(Amount) AS TotalRevenue
FROM dbo.Sales
GROUP BY ProductID;
```

Key Differences

- Data is **physically stored**
- Faster read performance
- Requires SCHEMABINDING
- Many creation restrictions

Real-Life Use Cases

- **E-commerce:** Total sales per product for dashboards
- **Banking:** Daily transaction summaries
- **Business intelligence:** Reports with heavy aggregations

Limitations & Performance Considerations

- Increased storage usage
- Slower INSERT, UPDATE, DELETE on base tables
- Many SQL restrictions (no LEFT JOIN, DISTINCT, UNION)
- More maintenance overhead

3. Partitioned View (Union View)

What is it?

A **Partitioned View** combines multiple tables using UNION ALL.
It is often used when large tables are split into smaller ones (horizontal partitioning).

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

Key Differences

- Combines data from multiple tables
- Useful for large or distributed datasets
- Tables must have the same structure

Real-Life Use Cases

- **Banking:** Transactions stored by year
- **E-commerce:** Orders split by region or year
- **Telecom:** Call records stored monthly

Limitations & Performance Considerations

- All tables must have identical columns and data types
- Complex DML operations
- Query performance depends on partition design
- Requires CHECK constraints for optimization

View Type	Storage	Best Use Case
Standard View	Virtual	Security & simplicity

Indexed View	Physical	Fast reporting & aggregation
Partitioned View	Virtual	Large, segmented datasets

1. Which Types of Views Allow DML Operations?

Updatable Views
These are views on which you *can* perform INSERT, UPDATE, and DELETE. In general:

- The view must be based on a **single base table** (no ambiguity).
- It must not include **aggregate functions** (like SUM, AVG), GROUP BY, DISTINCT, or set operations (UNION, etc.).
- It should include **primary key columns** and all **NOT NULL** columns of the base table so that inserts can supply required data.
- Columns in the view must map directly to real columns — no computed/derived expressions.Oracle Docs+1

With Check Option Views
If you define a view with WITH CHECK OPTION, the database *enforces* the view's WHERE clause when you do DML — meaning that inserted/updated rows must still satisfy the view's conditions.Oracle Docs

Views with INSTEAD OF Triggers
Some systems (like Oracle or SQL Server) let you define INSTEAD OF triggers on views — these intercept DML and let you write custom logic to update one or more base tables even when the view itself wouldn't normally be updatable directly. This effectively lets you support DML on more complex views (e.g., joins).Oracle Docs

Not Updatable / Read-Only Views:
Views that contain:

- GROUP BY, HAVING, DISTINCT
- Aggregate functions
- Joins on multiple tables (unless handled with triggers)
- Set operators (UNION, INTERSECT, etc.)
- Derived columns/expressions

are generally **not updatable**, meaning DML won't work directly.

⚠ 2. Restrictions When Performing DML on Views

Even when DML *is* allowed, there are important limitations:

◇ View Must Map Clearly to Base Table

The database must be able to trace your changes on the view back to a specific row in a base table. If a column is computed (salary * 1.1) or from an aggregation (avg(salary)), the system cannot do that reliably.

◇ Multi-Table Views Are Harder

Without triggers, views based on joins usually aren't updatable.
Even if a view *contains* more than one table, you can often only modify data in **one underlying table** at a time through it.

◇ With Check Option Enforcement

If the view has a filtering WHERE clause (WHERE status = 'Active') with WITH CHECK OPTION, any inserted/updated rows must still satisfy that filter. Otherwise, the DBMS rejects the DML.

◇ Base Table Rules Still Apply

Constraints like foreign keys, NOT NULL, and uniqueness are still enforced just as if the DML was run on the base table itself.

3. Real-Life Example Where Updating a View Is Useful

Example: E-Commerce Order Management

Suppose you have an online store with a table Orders:

```
CREATE TABLE Orders (
    OrderID INT PRIMARY KEY,
    CustomerID INT,
    OrderStatus VARCHAR(20),
    TotalAmt DECIMAL(10,2),
    ProcessedAt DATETIME
);
```

Now support staff can:

- **Update the status** of an order when resolved:

```
UPDATE OpenOrders
SET OrderStatus = 'Cancelled'
WHERE OrderID = 12345;
```

Delete orders that are invalid or duplicates:

```
DELETE FROM OpenOrders
WHERE OrderID = 67890;
```

Because the view filters only open orders and has WITH CHECK OPTION, any change that would move data *outside* that set (e.g., marking an order as 'Returned') won't be allowed through this view unless it still matches WHERE OrderStatus = 'Open'. This helps **enforce business rules while limiting visibility**.

💡 This pattern lets you give users the ability to modify underlying data *only in appropriate contexts*, without granting full access to the base table itself — great for roles like support, auditing, or restricted admin tools.

Summary

Yes — you *can* use DML (INSERT/UPDATE/DELETE) on views, but only when:

✅ The view is updatable — typically based on a single table
✅ It doesn't involve aggregates, distinct, or complex expressions
✅ All necessary base columns (especially keys and NOT NULL fields) are present
✅ Optionally using WITH CHECK OPTION to enforce filters
✅ Or using triggers (INSTEAD OF) to explicitly handle updates on complex views

Otherwise, the view acts as read-only for data modifications.

How Views Simplify Complex Queries in SQL

1. Introduction

In real-world database systems, especially in domains like **banking**, queries often involve multiple tables joined together (Customers, Accounts, Transactions, etc.). Writing and maintaining these JOIN-heavy queries repeatedly can be time-consuming and error-prone.

A **View** helps simplify this complexity by storing a predefined query that can be reused like a virtual table.

2. How Views Simplify JOIN-Heavy Queries

A SQL View simplifies complex queries in the following ways:

☑ 1. Hides Complexity

- Instead of writing long JOIN statements every time, users can query a view as if it were a single table.
- The JOIN logic is written **once** and reused.

☑ 2. Improves Readability

- Queries become shorter, cleaner, and easier to understand.
- This is especially helpful for non-technical users (e.g., call center agents, analysts).

☑ 3. Ensures Consistency

- Everyone uses the same logic for joins and calculations.

- Prevents mistakes caused by writing joins differently each time.

☑ 4. Enhances Security

- Users can be given access to the view without accessing the underlying tables directly.

3. Banking Scenario: Call Center Account Summary

📌 Real-Life Use Case

In a **banking system**, call center agents frequently need to view:

- Customer name
- Account number
- Account type
- Current balance

Instead of writing complex JOIN queries every time, we can create a **View** that provides an **account summary**.

4. Tables Used (Assumption)

Customer Table

CustomerID	CustomerName	Phone

Account Table

| AccountID | CustomerID | AccountType | Balance |

5. Complex Query WITHOUT a View:

```sql
SELECT
    c.CustomerID,
    c.CustomerName,
    a.AccountID,
    a.AccountType,
    a.Balance
FROM Customer c
JOIN Account a
ON c.CustomerID = a.CustomerID;
```

🔴 **Problem:**

- This query must be rewritten every time the data is needed.
- Repetition increases errors and reduces productivity.

6. Creating a View (Customer + Account)

```sql
CREATE VIEW vw_CustomerAccountSummary
AS
SELECT
    c.CustomerID,
    c.CustomerName,
    a.AccountID,
    a.AccountType,
    a.Balance
FROM Customer c
JOIN Account a
ON c.CustomerID = a.CustomerID;
```

✓ The JOIN logic is now saved permanently in the database

7. Using the View (Simplified Query)

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

🟢 **Result**:

- No JOIN required
- Faster, cleaner, and easier to use

8. Benefits for Call Center Agents

Without View	With View
Long JOIN queries	Simple SELECT
Hard to understand	Easy to read
Repeated logic	Centralized logic
Higher error risk	Consistent results

9. Conclusion

Views are a powerful tool for simplifying complex SQL queries. In banking systems, they:

- Reduce query complexity
- Improve productivity
- Ensure consistent and secure access to data

By using views such as **Customer Account Summary**, teams like call center agents can retrieve critical information quickly without needing advanced SQL knowledge.