SQL Server Views

Introduction

In modern database management systems, **views** play a crucial role in simplifying data access, enhancing security, and improving performance in certain scenarios. A view is a virtual table based on a SQL query, which allows users to access data without directly interacting with the base tables. This report provides an in-depth analysis of the different types of views in SQL Server, their characteristics, use cases, limitations, and the ability to perform DML operations on them. Additionally, the report demonstrates how views can simplify complex queries in real-world applications such as banking, e-commerce, and university systems.

1. Types of Views in SQL Server

1.1 Standard View

A **Standard View** is a virtual table created using a SELECT statement. It does not store data physically; it only contains the definition of the SQL query. When queried, the data is dynamically retrieved from the underlying tables. Standard Views are primarily used to simplify complex queries, promote query reuse, and hide sensitive information from users.

Key Differences:

Standard Views differ from Indexed and Partitioned Views in that they **do not store data** and do not inherently improve query performance. They are the simplest type of view and provide flexibility for various reporting and data access needs.

Real-life Use Cases:

- **University Systems:** Displaying active students in a semester while hiding sensitive personal data.
- **E-commerce Platforms:** Displaying only orders ready for shipment without exposing all order details.

Limitations & Performance Considerations:

- ORDER BY cannot be used directly in the view definition unless combined with TOP.

- Complex queries within a view may affect performance on large datasets.
- Standard Views do not provide performance gains over querying base tables directly.

1.2 Indexed View

- An **Indexed View** is a view with a **clustered index** that stores the query results physically on disk. This makes queries, especially those involving **aggregations or complex calculations**, faster since the results are precomputed and maintained automatically.
- **Key Differences:**
  Indexed Views differ from Standard Views in that they **improve query performance** but consume additional storage and require strict design rules. They can impact `INSERT`, `UPDATE`, and `DELETE` operations on underlying tables due to index maintenance. Design constraints include the use of `SCHEMABINDING` and avoiding unsupported constructs like `DISTINCT` or `OUTER JOIN`.
- **Real-life Use Cases:**
- **Banking Systems:** Calculating daily total transactions per branch for quick reporting.
- **Data Warehousing:** Frequently accessed summarized data for dashboards and analytics.
- **Limitations & Performance Considerations:**
- Slower write operations on underlying tables.
- Requires additional storage space.
- Must follow strict design rules to maintain index integrity.

1.3 Partitioned View (Union View)

A **Partitioned View**, also known as a **Union View**, combines multiple similar tables using `UNION ALL` to appear as a single logical table. Each table typically contains a subset of the data, such as orders for a specific year or transactions by region, allowing large datasets to be managed efficiently while still providing unified access.

Key Differences:
Partitioned Views work across multiple tables and support **partition elimination**, improving query performance for large datasets. All tables must have identical structures and proper CHECK constraints to maintain data integrity.

Real-life Use Cases:

- **E-commerce Systems:** Orders stored in separate tables per year, queried as one table.
- **Large Enterprises:** Historical records distributed across multiple tables for efficient access.

Limitations & Performance Considerations:

- Complex to maintain compared to other views.
- Requires careful planning of table structures and constraints.
- Partition elimination depends on proper configuration.

2. DML Operations on Views

Which Views Allow DML Operations?

- **Standard Views:** Allowed, but with restrictions.
- **Indexed Views:** Limited, possible under specific conditions.
- **Partitioned Views:** Usually not allowed for DML.

Restrictions & Limitations

For DML operations such as `INSERT`, `UPDATE`, and `DELETE`:

- The view must reference a **single base table**.
- Cannot include `GROUP BY`, `DISTINCT`, aggregate functions, or joins (for updates/inserts).
- Indexed Views require unique clustered indexes and deterministic expressions for DML operations.

Real-life Example

SQLQuery1.sql - S...(SHIMA\shima (69))*

```
create database empl
use empl

CREATE TABLE Employees (
     EmployeeID INT PRIMARY KEY IDENTITY(1,1),
     Name NVARCHAR(50) NOT NULL,
     Phone NVARCHAR(15),
     Email NVARCHAR(50),
     Salary DECIMAL(10,2)
)
INSERT INTO Employees (Name, Phone, Email, Salary)
VALUES
('Alice Johnson', '12345678', 'alice@example.com', 5000),
('Bob Smith', '87654321', 'bob@example.com', 4500),
('Charlie Lee', '55555555', 'charlie@example.com', 6000)

CREATE VIEW vw_EmployeeContacts
AS
SELECT EmployeeID, Name, Phone
FROM Employees

UPDATE vw_EmployeeContacts
SET Phone = '99999999'
WHERE EmployeeID = 2

INSERT INTO vw_EmployeeContacts (Name, Phone)
VALUES ('David Green', '44444444')

DELETE FROM vw_EmployeeContacts
WHERE EmployeeID = 3

select * from vw_EmployeeContacts
```

108 %

Results | Messages

	EmployeeID	Name	Phone
1	1	Alice Johnson	12345678
2	2	Bob Smith	99999999
3	4	David Green	44444444

Benefits:

- **Reduces development time**
- **Minimizes errors in repetitive queries**

- **Simplifies maintenance for frequently accessed data**

Conclusion

Views in SQL Server are powerful tools that **simplify data access, enhance security, and improve performance** when used appropriately. Standard Views provide flexibility and simplicity, Indexed Views improve read/query performance for aggregated data, and Partitioned Views enable efficient management of large datasets across multiple tables. Understanding the characteristics, limitations, and real-life applications of each type ensures optimal use in systems ranging from banking and e-commerce to university management.

How Views Simplify Complex Queries

In modern database systems, many queries involve joining multiple tables to retrieve meaningful information. Writing such **JOIN-heavy queries repeatedly** can lead to several challenges: it is time-consuming, prone to errors, and difficult to maintain. Each time a query is needed, developers or analysts must carefully write and test multiple joins, which increases the risk of mistakes and inconsistencies.

A **view** provides a solution by encapsulating the complexity of these queries into a **virtual table**. Once a view is created, users can query it as if it were a regular table, without needing to rewrite the complex joins. This abstraction simplifies data access and promotes consistency across the system.

The benefits of using views for complex queries include:

1. **Simplified Data Access:** Users can retrieve information without knowing the underlying table structures or writing long SQL queries.
2. **Query Standardization:** Frequently used queries are stored in one place, ensuring consistency and reducing the risk of errors.
3. **Error Reduction:** By avoiding repeated coding of complex joins, the chance of introducing mistakes is minimized.
4. **Data Security:** Views can expose only the necessary columns, hiding sensitive data from unauthorized users.

5. **Maintenance Efficiency:** Changes in the underlying table structure can be addressed in the view definition, minimizing the need to update multiple queries in the application or reports.

Applied Example: Using Views to Simplify JOIN-heavy Queries

In banking systems, users frequently need to access combined information from multiple tables, such as **customer details and account information**, or **accounts and their transactions**. Writing these JOIN-heavy queries repeatedly can be cumbersome, time-consuming, and error-prone. Views provide an effective solution by encapsulating these complex queries into a single, reusable virtual table.

Example 1: Customer + Account

Suppose we have two tables: `Customers` and `Accounts`. A common query retrieves all accounts for each customer:

```
CREATE VIEW vw_CustomerAccounts

AS

SELECT c.CustomerID, c.CustomerName, a.AccountNumber, a.AccountType, a.Balance

FROM Customers c

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

Now, instead of rewriting the JOIN every time, a simple query like the following suffices:

```
SELECT *

FROM vw_CustomerAccounts

WHERE Balance > 1000;
```

This allows call center agents to quickly get account summaries for customers with balances over a certain threshold.

Example 2: Account + Transaction

To provide detailed transaction information for each account, we can join `Accounts` and `Transactions`:

```
CREATE VIEW vw_AccountTransactions

AS

SELECT a.AccountNumber, a.CustomerID, t.TransactionID, t.TransactionDate, t.Amount, t.TransactionType

FROM Accounts a

JOIN Transactions t ON a.AccountNumber = t.AccountNumber;
```

Querying this view becomes straightforward:

```
SELECT *

FROM vw_AccountTransactions

WHERE TransactionDate >= '2025-01-01';
```

Benefits

1. **Simplified Queries:** Users no longer need to write complex JOINs repeatedly.
2. **Consistency:** Ensures the same query logic is applied across different teams and reports.
3. **Error Reduction:** Minimizes coding mistakes when accessing complex datasets.
4. **Efficiency for Frequent Tasks:** Staff such as call center agents can retrieve account summaries quickly without constructing queries manually.

Part 2: Real-Life Implementation Task (Banking System)& Part 3: View Creation Scenarios

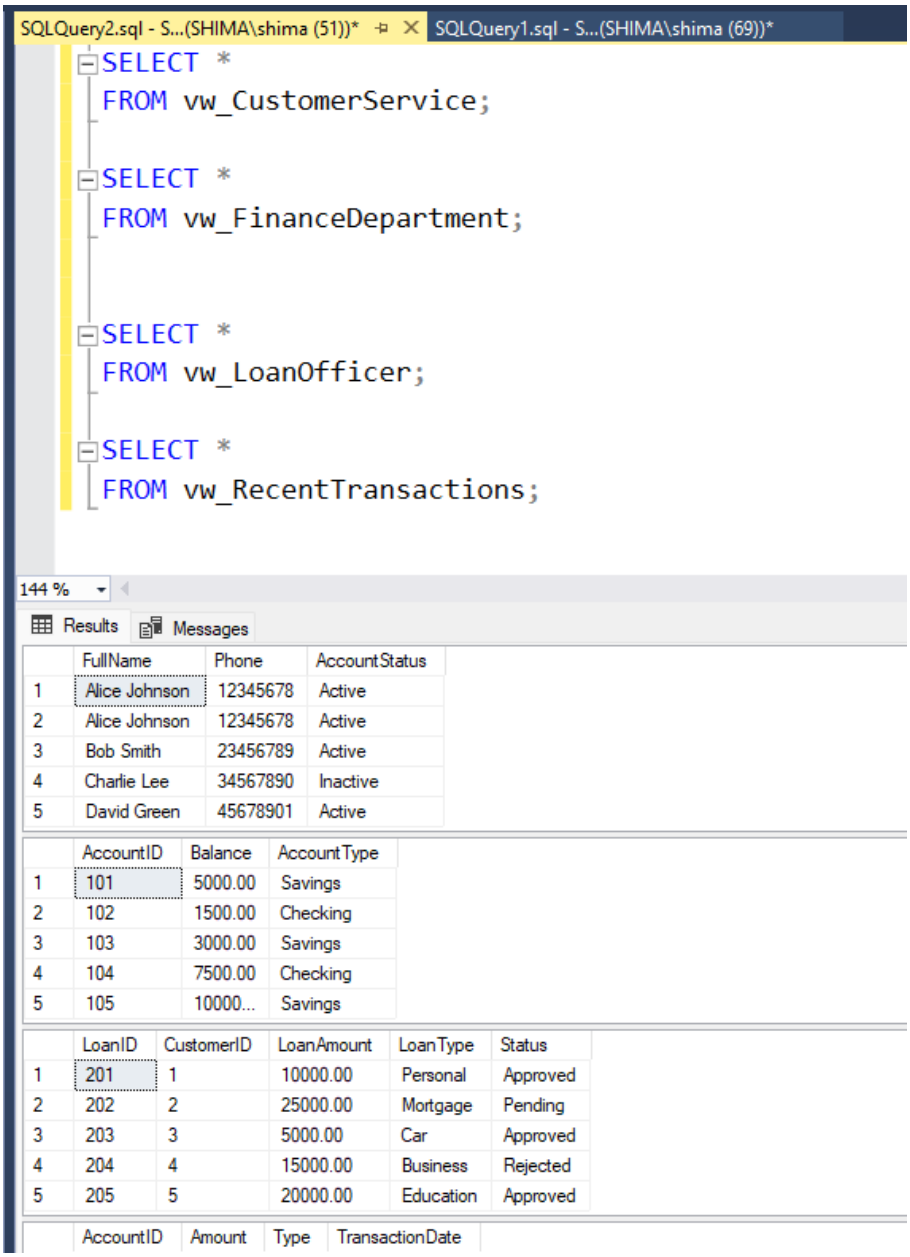
SQLQuery2.sql - S...(SHIMA\shima (51))* | SQLQuery1.sql - S...(SHIMA\shima (69))*

```sql
create database bankdata
use bankdata
CREATE TABLE Customer (
CustomerID INT PRIMARY KEY,
FullName NVARCHAR(100),
Email NVARCHAR(100),
Phone NVARCHAR(15),
SSN CHAR(9)
)
CREATE TABLE Account (
    AccountID INT PRIMARY KEY,
    CustomerID INT FOREIGN KEY REFERENCES Customer(CustomerID),
    Balance DECIMAL(10, 2),
    AccountType VARCHAR(50),
    Status VARCHAR(20)
)

CREATE TABLE Transactions (
    TransactionID INT PRIMARY KEY,
    AccountID INT FOREIGN KEY REFERENCES Account(AccountID),
    Amount DECIMAL(10, 2),
    Type VARCHAR(10), -- Deposit, Withdraw
    TransactionDate DATETIME
)
CREATE TABLE Loan (
    LoanID INT PRIMARY KEY,
    CustomerID INT FOREIGN KEY REFERENCES Customer(CustomerID),
    LoanAmount DECIMAL(12, 2),
    LoanType VARCHAR(50),
    Status VARCHAR(20)
)
iNSERT INTO Customer (CustomerID, FullName, Email, Phone, SSN)
VALUES
(1, 'Alice Johnson', 'alice.johnson@example.com', '12345678', '111223333'),
(2, 'Bob Smith', 'bob.smith@example.com', '23456789', '222334444'),
(3, 'Charlie Lee', 'charlie.lee@example.com', '34567890', '333445555'),
(4, 'David Green', 'david.green@example.com', '45678901', '444556666'),
(5, 'Eva Brown', 'eva.brown@example.com', '56789012', '555667777');
```

SQLQuery2.sql - S...(SHIMA\shima (51))* | SQLQuery1.sql - S...(SHIMA\shima (69))*

```sql
INSERT INTO Account (AccountID, CustomerID, Balance, AccountType, Status)
VALUES
(101, 1, 5000.00, 'Savings', 'Active'),
(102, 1, 1500.00, 'Checking', 'Active'),
(103, 2, 3000.00, 'Savings', 'Active'),
(104, 3, 7500.00, 'Checking', 'Inactive'),
(105, 4, 10000.00, 'Savings', 'Active');

INSERT INTO Transactions (TransactionID, AccountID, Amount, Type, TransactionDate)
VALUES
(1001, 101, 2000.00, 'Deposit', '2025-01-10'),
(1002, 101, 500.00, 'Withdraw', '2025-01-15'),
(1003, 102, 1500.00, 'Deposit', '2025-01-12'),
(1004, 103, 1000.00, 'Withdraw', '2025-01-20'),
(1005, 105, 5000.00, 'Deposit', '2025-01-25');

INSERT INTO Loan (LoanID, CustomerID, LoanAmount, LoanType, Status)
VALUES
(201, 1, 10000.00, 'Personal', 'Approved'),
(202, 2, 25000.00, 'Mortgage', 'Pending'),
(203, 3, 5000.00, 'Car', 'Approved'),
(204, 4, 15000.00, 'Business', 'Rejected'),
(205, 5, 20000.00, 'Education', 'Approved');

----------------------------------------------------Part 3: View Creation Scenarios

 CREATE VIEW vw_CustomerService
AS
SELECT
    c.FullName,
    c.Phone,
    a.Status AS AccountStatus
FROM Customer c
JOIN Account a ON c.CustomerID = a.CustomerID

CREATE VIEW vw_FinanceDepartment
AS
SELECT
    AccountID,
    Balance,
```

```sql
CREATE VIEW vw_LoanOfficer
AS
SELECT
    LoanID,
    CustomerID,
    LoanAmount,
    LoanType,
    Status
FROM Loan

CREATE VIEW vw_RecentTransactions
AS
SELECT
    AccountID,
    Amount,
    Type,
    TransactionDate
FROM Transactions
WHERE TransactionDate >= DATEADD(DAY, -30, GETDATE())

SELECT *
FROM vw_CustomerService;

SELECT AccountID, Balance, AccountType
FROM vw_FinanceDepartment
WHERE Balance > 3000;

SELECT *
FROM vw_LoanOfficer
WHERE Status = 'Approved';

SELECT *
FROM vw_RecentTransactions
ORDER BY TransactionDate DESC;
```



	FullName	Phone	AccountStatus
1	Alice Johnson	12345678	Active
2	Alice Johnson	12345678	Active
3	Bob Smith	23456789	Active
4	Charlie Lee	34567890	Inactive
5	David Green	45678901	Active

	AccountID	Balance	AccountType
1	101	5000.00	Savings
2	102	1500.00	Checking
3	103	3000.00	Savings
4	104	7500.00	Checking
5	105	10000...	Savings

	LoanID	CustomerID	LoanAmount	LoanType	Status
1	201	1	10000.00	Personal	Approved
2	202	2	25000.00	Mortgage	Pending
3	203	3	5000.00	Car	Approved
4	204	4	15000.00	Business	Rejected
5	205	5	20000.00	Education	Approved

	AccountID	Amount	Type	TransactionDate