Part 1: Research & Documentation

1. Standard View

Attribute	Description
Definition	A virtual table based on a SELECT query; does not store data physically.
Key Differences	Does not store data physically; simpler than Indexed View; cannot span multiple tables like Partitioned View.
Real-life Use Cases	Banking: show customer names with masked account numbers. E-commerce: display products with categories. University: list student names with enrolled courses.
Limitations & Performance	Slower on large datasets because data is fetched at runtime. Cannot have indexes. Some insert/update/delete operations may be restricted.

2. Indexed View

Attribute	Description
Definition	A view stored physically on disk with a unique clustered index for faster query performance.
Key Differences	Stores data physically; auto-updates when base tables change; better performance for complex queries.
Real-life Use Cases	Banking: pre-calculated account balances for dashboards. E-commerce: daily sales totals. University: pre-calculated average grades per course.
Limitations & Performance	Slower inserts/updates due to index maintenance. Requires more storage. Strict requirements on determinism, schema binding, and functions used.

3. Partitioned View

Attribute	Description
Definition	Combines data from multiple tables using `UNION ALL` to manage large datasets efficiently.
Key Differences	Designed for large datasets; can span multiple tables or databases; useful for horizontal partitioning.
Real-life Use Cases	Banking: combine transaction tables per branch. E-commerce: combine sales from multiple regional warehouses. University: merge student records from different departments.
Limitations & Performance	Maintenance can be complex if partitions grow. Performance depends on querying the correct partitions. Requires careful indexing and partition strategy.

- ## 1. Can We Use DML (INSERT, UPDATE, DELETE) on Views?

Yes, DML operations (INSERT, UPDATE, DELETE) can be performed on views, but only under certain conditions. Views in SQL Server are **virtual tables**: some store only the query definition (standard views), while others may store precomputed data (indexed views) or combine multiple tables (partitioned views).

- **Standard views** can be updatable if they map directly to a single base table and do not contain complex expressions.
- **Indexed views** generally do not allow direct DML; changes should be made on the base table.
- **Partitioned (union) views** can support DML, but only if each underlying table is properly constrained, and the operation affects a single table.

2. Which types of views allow DML operations?

- **Standard Views:**
  Can allow INSERT, UPDATE, and DELETE if the view references a single base table and avoids joins, GROUP BY, DISTINCT, or aggregate functions.
- **Partitioned (Union) Views:**
  Can allow DML, but only if **check constraints** exist on all underlying

tables to determine where the row belongs. The operation must affect only **one base table at a time**.

- **Indexed Views:**
  Typically do not allow direct DML. Any insert, update, or delete should be done on the underlying table, which automatically maintains the indexed view.

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

A. Standard Views:

- Cannot perform DML if the view has **joins** between multiple tables.
- Cannot update **computed columns** or columns derived from functions or expressions.
- Cannot insert data that violates constraints in the base table.
- Cannot modify rows that **do not exist** in the base table or that fail the view's WHERE condition.

B. Partitioned Views:

- Each table must have a **check constraint** to identify which table should store a new row.
- Only **one underlying table** can be modified at a time.
- If constraints are missing or violated, SQL Server may reject the operation.

C. Indexed Views:

- The view must be **schema-bound** to allow certain DML operations indirectly.
- Cannot include forbidden expressions like `GETDATE()` or `ROW_NUMBER()` in indexed views.
- Direct DML on the indexed view itself is rarely done; it is maintained automatically when the base tables change.

Performance Considerations:

- DML on updatable views can be slower if the view has a **complex WHERE clause** or joins because SQL Server must process the query to determine the rows to update.
- Partitioned views require proper constraints to avoid scanning all underlying tables.

4. Give at least one real-life example where updating a view is useful

SQLQuery2.sql -...MANAR\maaa7 (66))*

```sql
create database emp
use emp

CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,
    FirstName NVARCHAR(50),
    LastName NVARCHAR(50),
    Phone NVARCHAR(20),
    JobTitle NVARCHAR(50),
    Status NVARCHAR(10)
)

INSERT INTO Employees VALUES
(101, 'John', 'Doe', '111-111-1111', 'Software Engineer', 'Active'),
(102, 'Jane', 'Smith', '222-222-2222', 'HR Manager', 'Active'),
(103, 'Alice', 'Brown', '333-333-3333', 'Intern', 'Inactive')

select * from Employees
-- Create a view to show only active employees
CREATE VIEW ActiveEmployees AS
SELECT EmployeeID, FirstName, LastName, Phone, JobTitle
FROM Employees
WHERE Status = 'Active'
-- Update phone number for an employee using the view
UPDATE ActiveEmployees
SET Phone = '123-456-7890'
WHERE EmployeeID = 101

select * from Employees
```

119 %

Results | Messages

	EmployeeID	FirstName	LastName	Phone	JobTitle	Status
1	101	John	Doe	123-456-7890	Software Engineer	Active
2	102	Jane	Smith	222-222-2222	HR Manager	Active
3	103	Alice	Brown	333-333-3333	Intern	Inactive

2. How Can Views Simplify Complex Queries?

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

Views in SQL Server act as **virtual tables** created from stored SELECT queries. They are extremely useful for simplifying queries that involve **multiple table joins**, filters, and calculations.

When a query requires joining several tables, such as `Customer`, `Account`, and `Transaction` in a banking system, writing the full SQL statement every time can be **time-consuming, repetitive, and error-prone**. Views solve this problem by **encapsulating the complex query logic** into a single object.

By using a view, users can access the same combined data with a **simple SELECT statement**, without worrying about the underlying joins or filtering logic. This makes queries **shorter, easier to read, and maintainable**, while ensuring consistency in business rules.

- **Practical example:**
  In a banking call center, agents frequently need to see customer account summaries and recent transactions. Without a view, they would have to repeatedly write complex JOIN queries. By creating a view that joins `Customer` + `Account` or `Account` + `Transaction`, agents can simply query the view to get all relevant information quickly.

-- Create Customer + Account View

```
CREATE VIEW CustomerAccountView AS
SELECT
    C.CustomerID,
    C.FirstName,
    C.LastName,
    C.Phone,
    A.AccountID,
    A.AccountType,
    A.Balance
FROM Customer C
JOIN Account A ON C.CustomerID = A.CustomerID;
```

-- Example usage

SELECT * FROM CustomerAccountView

-- Create Account + Transaction View

```
CREATE VIEW AccountTransactionView AS
SELECT
    A.AccountID,
    A.AccountType,
    A.Balance,
    T.TransactionID,
    T.TransactionDate,
    T.Amount,
    T.Type
FROM Account A
LEFT JOIN Transaction T ON A.AccountID = T.AccountID
```

-- Example usage

SELECT * FROM AccountTransactionView

WHERE AccountID = 201

- **How Using Views Reduces the Need to Repeat Long Queries**

In SQL Server, queries that involve **multiple tables with joins** can become very long and complicated. For example, retrieving customer account information along with transaction history may require joining `Customer`, `Account`, and `Transaction` tables with multiple conditions and filters. Writing this same complex query every time is **time-consuming, error-prone, and difficult to read**.

By creating a **view**, the complex query is stored once in the database as a virtual table. Users can then query the view with a simple SELECT statement instead of repeating the entire JOIN logic each time.

Benefits of using a view for repetitive queries:

1. **Shorter queries:** Instead of rewriting multiple JOINs, users can write a simple query like `SELECT * FROM CustomerAccountView`.
2. **Consistency:** The view ensures all users access the data in the same way, applying the same joins and filters.
3. **Reduced errors:** Users don't need to remember the exact JOIN conditions, reducing mistakes.
4. **Easy maintenance:** If table structure or business rules change, updating the view automatically updates all queries using it.

Part 2: Real-Life Implementation Task (Banking System)

```sql
INSERT INTO Customer (CustomerID, FullName, Email, Phone, SSN) VALUES
(101, 'John Doe', 'john.doe@email.com', '111-111-1111', '123456789'),
(102, 'Jane Smith', 'jane.smith@email.com', '222-222-2222', '987654321'),
(103, 'Alice Brown', 'alice.brown@email.com', '333-333-3333', '456789123')
select * from Customer

INSERT INTO Account (AccountID, CustomerID, Balance, AccountType, Status) VALUES
(201, 101, 5000.00, 'Savings', 'Active'),
(202, 101, 1500.00, 'Checking', 'Active'),
(203, 102, 3000.00, 'Savings', 'Active'),
(204, 103, 1000.00, 'Checking', 'Inactive')
select * from Account

INSERT INTO Transactions (TransactionID, AccountID, Amount, Type, TransactionDate) VALUES
(301, 201, 1000.00, 'Deposit', '2025-12-01 10:00:00'),
(302, 201, 200.00, 'Withdraw', '2025-12-05 12:30:00'),
(303, 202, 500.00, 'Deposit', '2025-12-03 14:00:00'),
(304, 203, 300.00, 'Deposit', '2025-12-02 09:00:00')
select * from Transactions

INSERT INTO Loan (LoanID, CustomerID, LoanAmount, LoanType, Status) VALUES
(401, 101, 10000.00, 'Personal', 'Active'),
(402, 102, 50000.00, 'Home', 'Active'),
(403, 103, 2000.00, 'Auto', 'Closed')
select * from Loan
```

Part 3: View Creation Scenarios

```sql
-- Create Customer Service View
CREATE VIEW CustomerServiceView AS
SELECT
    C.FullName,
    C.Phone,
    A.Status AS AccountStatus
FROM Customer C
JOIN Account A ON C.CustomerID = A.CustomerID
SELECT * FROM CustomerServiceView
```

108 %

Results | Messages

	FullName	Phone	AccountStatus
1	John Doe	111-111-1111	Active
2	John Doe	111-111-1111	Active
3	Jane Smith	222-222-2222	Active
4	Alice Brown	333-333-3333	Inactive

```sql
-- Create Finance Department View
CREATE VIEW FinanceDeptView AS
SELECT
    AccountID,
    Balance,
    AccountType
FROM Account
SELECT * FROM FinanceDeptView
```

108 %

Results | Messages

	AccountID	Balance	AccountType
1	201	5000.00	Savings
2	202	1500.00	Checking
3	203	3000.00	Savings
4	204	1000.00	Checking

```sql
-- Create Loan Officer View
CREATE VIEW LoanOfficerView AS
SELECT
    LoanID,
    CustomerID,
    LoanAmount,
    LoanType,
    Status AS LoanStatus
FROM Loan
SELECT * FROM LoanOfficerView
```

108 %

Results | Messages

	LoanID	CustomerID	LoanAmount	LoanType	LoanStatus
1	401	101	10000.00	Personal	Active
2	402	102	50000.00	Home	Active
3	403	103	2000.00	Auto	Closed

```sql
-- Create Transaction Summary View (last 30 days)
CREATE VIEW TransactionSummaryView AS
SELECT
    AccountID,
    Amount,
    TransactionDate,
    Type AS TransactionType
FROM Transactions
WHERE TransactionDate >= DATEADD(DAY, -30, GETDATE())
SELECT * FROM TransactionSummaryView
```

108 %

Results | Messages

	AccountID	Amount	TransactionDate	TransactionType
1	201	1000.00	2025-12-01 10:00:00.000	Deposit
2	201	200.00	2025-12-05 12:30:00.000	Withdraw
3	202	500.00	2025-12-03 14:00:00.000	Deposit
4	203	300.00	2025-12-02 09:00:00.000	Deposit