Views

Introduction To Views

A **View** in DBMS is a **virtual table** created using a SELECT query on one or more base tables. A view does not store data physically; instead, it stores the **definition of the query**. Whenever a view is accessed, the DBMS dynamically retrieves data from the underlying base tables.

Definition: > A view is a logical representation of data derived from one or more database tables.

Need for Views

Views are required in DBMS for the following reasons: - To simplify complex queries - To improve database security - To provide restricted access to sensitive data - To support data abstraction - To present customized views of data for different users

Syntax for Creating A View

```
CREATE VIEW view_name AS
SELECT column1, column2
FROM table_name
WHERE condition;
```

Example of View Creation

Base Table: STUDENT

RollNo	Name	Dept	Marks
101	Ravi	CSE	85
102	Anu	ECE	72
103	Kiran	CSE	90

Create a View for CSE Students

```
CREATE VIEW CSE_Students AS
SELECT RollNo, Name, Marks
FROM Student
WHERE Dept = 'CSE';
```

Resulting View (Virtual Table)

RollNo	Name	Marks
101	Ravi	85
103	Kiran	90

Types of Views

Simple View

- Created from a single table
- No GROUP BY, HAVING, or aggregate functions

```
CREATE VIEW Student_View AS
SELECT RollNo, Name
FROM Student;
```

Complex View

- Created from multiple tables
- Uses joins, GROUP BY, or aggregate functions

```
CREATE VIEW Dept_Avg_Marks AS
SELECT Dept, AVG(Marks) AS AvgMarks
FROM Student
GROUP BY Dept;
```

Materialized View

- Stores data physically
- Improves performance
- Requires refresh

```
CREATE MATERIALIZED VIEW Dept_View AS
SELECT Dept, COUNT(*) FROM Student GROUP BY Dept;
```

Read-Only View

- Data modification not allowed

```
CREATE VIEW ReadOnly_View AS
SELECT * FROM Student
WITH READ ONLY;
```

Updatable Views

A view is updatable if: - It is based on a single table - It does not contain DISTINCT, GROUP BY, HAVING - It does not use aggregate functions

Example

```
UPDATE Student_View
SET Name = 'Arjun'
WHERE RollNo = 101;
```

This updates the base table, not the view.

Dropping a View

```
DROP VIEW view_name;
```

Advantages of Views

- Enhanced security
- Logical data abstraction
- Simplified query execution
- Reusability of SQL queries
- Supports data independence

Limitations of Views

- Performance overhead for complex views
- Limited update capability
- Nested views increase complexity
- Materialized views consume storage

Difference between View and Table

Feature	View	Table
Data storage	Virtual	Physical
Memory usage	No	Yes
Security	High	Moderate
Updatable	Limited	Fully

Views and Data Security

Views restrict access to sensitive data by exposing only selected columns.

```
CREATE VIEW Faculty_View AS
SELECT RollNo, Name
FROM Student;
```

Views and Levels of Abstraction

Views belong to the **View Level (External Schema)** of DBMS architecture. They hide the complexity of the database and provide user-specific data representations.

Keys

Introduction

In a Database Management System (DBMS), a **key** is an attribute or a set of attributes that helps in uniquely identifying a tuple (row) in a relation (table). Keys play a crucial role in maintaining **data integrity**, establishing **relationships between tables**, and supporting **normalization**.

Importance of Keys

Keys are used to: - Uniquely identify records in a table - Prevent duplicate data - Establish relationships between tables - Enforce integrity constraints - Support normalization and efficient data retrieval

Types of Keys in DBMS

Super Key

A **super key** is any set of one or more attributes that can uniquely identify a record in a table.

Example: STUDENT(USN, Name, Email) - {USN} - {USN, Name} - {USN, Email}

All the above are super keys.

Candidate Key

A **candidate key** is a minimal super key, i.e., no proper subset of it can uniquely identify a record.

Example: - USN - Email

A table can have **multiple candidate keys**.

Primary Key

A **primary key** is one candidate key chosen to uniquely identify tuples in a table.

Characteristics: - Must be unique - Cannot contain NULL values - Only one primary key per table

Example: USN is chosen as the primary key in the STUDENT table.

Alternate Key

The candidate keys that are **not selected** as the primary key are called **alternate keys**.

Example: If USN is the primary key, then Email becomes an alternate key.

Composite Key

A **composite key** consists of two or more attributes used together to uniquely identify a record.

Example: ENROLLMENT(StudentID, CourseID)

Primary Key: (StudentID, CourseID)

Foreign Key

A **foreign key** is an attribute in one table that refers to the **primary key of another table**.

Purpose: - Maintains referential integrity

Example: STUDENT(DeptID) DEPARTMENT(DeptID → Primary Key)

Unique Key

A **unique key** ensures that all values in a column are unique.

Differences from Primary Key: - Can allow NULL values - A table can have multiple unique keys

Natural Key

A **natural key** is a real-world, meaningful attribute used to identify records.

Examples: - Aadhaar Number - Email ID

Surrogate Key

A **surrogate key** is an artificially generated key with no business meaning.

Example: - Auto-increment ID

Used widely for performance and simplicity.

Secondary Key

A **secondary key** is used for searching and indexing but does not uniquely identify records.

Example: DepartmentName

Comparison of Key Types

Key Type	Uniqueness	NULL Allowed	Count per Table
Super Key	Yes	No	Many

Key Type	Uniqueness	NULL Allowed	Count per Table
Candidate Key	Yes	No	Many
Primary Key	Yes	No	One
Alternate Key	Yes	No	Many
Foreign Key	No	Yes	Many
Unique Key	Yes	Yes	Many
Composite Key	Yes	No	One
Secondary Key	No	Yes	Many

SQL Example

```sql
CREATE TABLE Student (
    USN VARCHAR(10) PRIMARY KEY,
    Name VARCHAR(50),
    Email VARCHAR(50) UNIQUE,
    DeptID INT,
    FOREIGN KEY (DeptID) REFERENCES Department(DeptID)
);
```

DCL and TCL

Introduction

In a Database Management System (DBMS), SQL commands are categorized based on their functionality. Among these, **Data Control Language (DCL)** and **Transaction Control Language (TCL)** play a crucial role in **database security** and **transaction management** respectively. While DCL controls access to data, TCL manages changes made by transactions.

Data Control Language (DCL)

Definition

Data Control Language (DCL) consists of SQL commands used to **control access and permissions** on database objects such as tables, views, and procedures. DCL ensures database security by allowing or restricting user privileges.

DCL Commands

GRANT

The **GRANT** command is used to give specific privileges to users or roles.

Syntax:

```sql
GRANT privilege_name
ON object_name
TO user_name;
```

Example:

```sql
GRANT SELECT, INSERT
ON Student
TO user1;
```

Explanation: User `user1` is allowed to SELECT and INSERT data into the Student table.

REVOKE

The **REVOKE** command is used to remove previously granted privileges.

Syntax:

```
REVOKE privilege_name
ON object_name
FROM user_name;
```

Example:

```
REVOKE INSERT
ON Student
FROM user1;
```

Common DCL Privileges

- SELECT
- INSERT
- UPDATE
- DELETE
- REFERENCES
- ALL

Transaction Control Language (TCL)

Definition

Transaction Control Language (TCL) commands are used to manage **transactions** in a database. A transaction is a sequence of SQL operations executed as a single logical unit of work.

TCL Commands

COMMIT

The **COMMIT** command saves all changes made by the current transaction permanently to the database.

Syntax:

```
COMMIT;
```

ROLLBACK

The **ROLLBACK** command undoes changes made by the current transaction.

Syntax:

```
ROLLBACK;
```

SAVEPOINT

The **SAVEPOINT** command creates a point within a transaction to which you can later roll back.

Syntax:

```
SAVEPOINT savepoint_name;
```

Example:

```
SAVEPOINT sp1;
```

ROLLBACK TO SAVEPOINT

Used to roll back the transaction to a specific savepoint.

```
ROLLBACK TO sp1;
```

Example of Transaction Using TCL

```
INSERT INTO Student VALUES (101, 'Ravi');
SAVEPOINT s1;
UPDATE Student SET Name = 'Ravi Kumar' WHERE ID = 101;
ROLLBACK TO s1;
COMMIT;
```

Difference between DCL and TCL

Aspect	DCL	TCL
Full Form	Data Control Language	Transaction Control Language
Purpose	Controls user access	Manages transactions
Commands	GRANT, REVOKE	COMMIT, ROLLBACK, SAVEPOINT
Focus	Security	Consistency

Relation with ACID Properties

- **Atomicity:** Ensured using ROLLBACK
- **Consistency:** Maintained by COMMIT
- **Isolation:** Handled by DBMS internally
- **Durability:** COMMIT ensures permanent storage

ACID Properties

Introduction

In a Database Management System (DBMS), a **transaction** is a sequence of operations performed as a single logical unit of work. To ensure reliability and correctness of transactions, DBMS follows a set of four fundamental properties known as **ACID properties**: **Atomicity, Consistency, Isolation, and Durability**.

What is a Transaction?

A transaction is a collection of database operations such as INSERT, UPDATE, DELETE, or SELECT that must be executed completely or not executed at all.

Example: - Transfer of money from one bank account to another - Online ticket booking

ACID Properties

Atomicity

Definition: Atomicity ensures that a transaction is treated as an **indivisible unit**. Either all operations of the transaction are executed successfully, or none of them are.

Explanation: If a transaction fails in the middle, all changes made by it are rolled back.

Example: While transferring ₹10,000 from Account A to Account B: - Debit from Account A - Credit to Account B

If credit fails, debit must also be undone.

Mechanism Used: - ROLLBACK - Undo logs

Consistency

Definition: Consistency ensures that a transaction brings the database from one **valid state to another valid state**, maintaining all integrity constraints.

Explanation: After a transaction completes, all rules such as primary key, foreign key, and domain constraints must be satisfied.

Example: A student cannot be assigned to a non-existent department.

Mechanism Used: - Integrity constraints - Database rules and triggers

Isolation

Definition: Isolation ensures that multiple transactions executing concurrently do not interfere with each other. The effect of concurrent execution should be the same as if transactions were executed sequentially.

Problems Without Isolation: - Dirty Read - Non-repeatable Read - Phantom Read

Mechanism Used: - Locking protocols - Isolation levels

Durability

Definition: Durability guarantees that once a transaction is committed, its changes are **permanently saved** in the database, even in case of system failure.

Explanation: Committed data will not be lost due to crashes or power failures.

Mechanism Used: - COMMIT - Redo logs - Stable storage

ACID Properties with Summary Table

Property	Meaning	Ensured By
Atomicity	All or nothing execution	Rollback, Undo logs
Consistency	Valid database state	Constraints, Rules
Isolation	No interference	Locks, Isolation levels
Durability	Permanent storage	Commit, Redo logs

Importance of ACID Properties

- Ensures reliable transaction processing

- Maintains data accuracy and integrity
- Prevents data corruption
- Essential for banking, reservation, and financial systems

ACID Properties and TCL

- **COMMIT** ensures Durability
- **ROLLBACK** ensures Atomicity
- **SAVEPOINT** helps partial rollback

Aggregate Operators

Introduction

In DBMS, **aggregate operators (aggregate functions)** are used to perform calculations on a set of values and return a **single summarized result**. They are widely used in data analysis, reporting, and decision-making queries.

Aggregate operators work on multiple rows and produce one output value.

Common Aggregate Operators

COUNT()

Definition: Returns the number of rows that satisfy a given condition.

Syntax:

```
SELECT COUNT(column_name) FROM table_name;
```

Example:

```
SELECT COUNT(*) FROM Student;
```

Returns the total number of students.

SUM()

Definition: Returns the total sum of values in a numeric column.

Syntax:

```
SELECT SUM(column_name) FROM table_name;
```

Example:

```
SELECT SUM(salary) FROM Employee;
```

Returns the total salary of all employees.

AVG()

Definition: Returns the average value of a numeric column.

Syntax:

```
SELECT AVG(column_name) FROM table_name;
```

Example:

```sql
SELECT AVG(marks) FROM Student;
```

Returns the average marks of students.

MIN()

Definition: Returns the minimum value from a column.

Syntax:

```sql
SELECT MIN(column_name) FROM table_name;
```

Example:

```sql
SELECT MIN(age) FROM Student;
```

Returns the youngest student's age.

MAX()

Definition: Returns the maximum value from a column.

Syntax:

```sql
SELECT MAX(column_name) FROM table_name;
```

Example:

```sql
SELECT MAX(salary) FROM Employee;
```

Returns the highest salary.

Aggregate Operators with GROUP BY

The **GROUP BY** clause is used to apply aggregate functions to groups of rows.

Example:

```sql
SELECT department, COUNT(*)
FROM Employee
GROUP BY department;
```

Returns the number of employees in each department.

Aggregate Operators with HAVING Clause

The **HAVING** clause is used to filter groups based on aggregate conditions.

Example:

```sql
SELECT department, AVG(salary)
FROM Employee
GROUP BY department
HAVING AVG(salary) > 50000;
```

Displays departments with average salary greater than 50,000.

Aggregate Operators and NULL Values

- Aggregate functions **ignore NULL values**
- COUNT(*) counts all rows including NULLs
- COUNT(column) ignores NULL values

Difference Between WHERE and HAVING

WHERE Clause	HAVING Clause
Filters rows	Filters groups
Used before GROUP BY	Used after GROUP BY
Cannot use aggregates	Uses aggregate functions

Advantages of Aggregate Operators

- Simplifies complex calculations
- Useful for reports and analytics
- Improves query efficiency
- Reduces data processing overhead

Nested Queries and Correlated Queries

Introduction

In SQL, a **query within another query** is called a **nested query** or **subquery**. Subqueries are powerful tools used to retrieve data based on the results of another query. A special type of subquery, known as a **correlated query**, depends on the outer query for its execution.

Nested Queries (Subqueries)

Definition

A **nested query** is a query written inside another SQL query. The inner query executes first, and its result is passed to the outer query.

Nested queries can appear in: - WHERE clause - SELECT clause - FROM clause - HAVING clause

Syntax of Nested Query

```
SELECT column_name
FROM table_name
WHERE column_name OPERATOR (
    SELECT column_name
    FROM table_name
    WHERE condition
);
```

Types of Nested Queries

a) Single-Row Subquery

Returns exactly one row.

Example: Find employees earning more than the average salary.

```sql
SELECT EmpName
FROM Employee
WHERE Salary > (
    SELECT AVG(Salary)
    FROM Employee
);
```

b) Multiple-Row Subquery

Returns more than one row.

Example: Find employees working in departments located in Hyderabad.

```sql
SELECT EmpName
FROM Employee
WHERE DeptID IN (
    SELECT DeptID
    FROM Department
    WHERE Location = 'Hyderabad'
);
```

Operators used: IN, ANY, ALL

c) Multiple-Column Subquery

Returns more than one column.

Example:

```sql
SELECT EmpName
FROM Employee
WHERE (Salary, DeptID) IN (
    SELECT Salary, DeptID
    FROM Employee
    WHERE Designation = 'Manager'
);
```

d) Subquery in SELECT Clause

Returns a value for each row.

```sql
SELECT EmpName,
       (SELECT DeptName
        FROM Department
        WHERE DeptID = E.DeptID) AS DeptName
FROM Employee E;
```

e) Subquery in FROM Clause (Derived Table)

The subquery acts as a temporary table.

```sql
SELECT DeptID, AvgSalary
FROM (
    SELECT DeptID, AVG(Salary) AS AvgSalary
    FROM Employee
    GROUP BY DeptID
) AS DeptAvg;
```

Correlated Queries

Definition

A **correlated query** is a subquery that uses values from the outer query. The inner query is executed **once for each row** processed by the outer query.

Syntax of Correlated Query

```
SELECT column_name
FROM table_name outer
WHERE column_name OPERATOR (
    SELECT column_name
    FROM table_name inner
    WHERE inner.column = outer.column
);
```

Example of Correlated Query

Example: Find employees who earn more than the average salary of their department.

```
SELECT EmpName
FROM Employee E
WHERE Salary > (
    SELECT AVG(Salary)
    FROM Employee
    WHERE DeptID = E.DeptID
);
```

EXISTS with Correlated Query

The **EXISTS** operator checks whether the subquery returns any rows.

```
SELECT EmpName
FROM Employee E
WHERE EXISTS (
    SELECT 1
    FROM Department D
    WHERE D.DeptID = E.DeptID
);
```

Nested Queries vs Correlated Queries

Aspect	Nested Query	Correlated Query
Dependency	Independent	Depends on outer query
Execution	Once	Once per outer row
Performance	Faster	Slower
Complexity	Simple	Complex

Advantages of Nested and Correlated Queries

- Easy to understand and write
- Useful for complex conditions
- Reduce need for temporary tables

Limitations

- Correlated queries are slower for large datasets
- Difficult to optimize
- Joins are often preferred for performance

Nested Queries vs Joins

Feature	Nested Queries	Joins
Performance	Lower	Higher
Readability	High	Moderate
Best Use	Conditional filtering	Large datasets

Procedures

Introduction to Stored Procedures

A **Stored Procedure** in DBMS is a **precompiled set of SQL statements** stored in the database and executed as a single unit. Procedures are used to perform repetitive tasks, enforce business logic, and improve performance.

Definition: > A stored procedure is a named collection of SQL statements that can accept parameters, perform operations, and return results.

Why Stored Procedures Are Needed

Stored procedures are used for: - Reusability of SQL code - Improved performance due to precompilation - Centralized business logic - Reduced network traffic - Improved security

Syntax of Stored Procedures

General Syntax

```
CREATE PROCEDURE procedure_name
AS
BEGIN
   SQL statements;
END;
```

Example of a Simple Stored Procedure

Table: STUDENT

RollNo	Name	Dept	Marks
101	Ravi	CSE	85
102	Anu	ECE	72

Procedure to Display All Students

```
CREATE PROCEDURE GetStudents
AS
BEGIN
   SELECT * FROM Student;
END;
```

Executing the Procedure

```
EXEC GetStudents;
```

Stored Procedures With Parameters

Example: Procedure with Input Parameter

```
CREATE PROCEDURE GetStudentByDept (@DeptName VARCHAR(20))
AS
BEGIN
   SELECT RollNo, Name, Marks
   FROM Student
   WHERE Dept = @DeptName;
END;
```

Execution

```
EXEC GetStudentByDept 'CSE';
```

Types of Parameters

- **IN**: Accepts input value
- **OUT**: Returns value
- **INOUT**: Accepts and returns value

Example with OUT Parameter

```
CREATE PROCEDURE GetTotalStudents (@Total INT OUTPUT)
AS
BEGIN
   SELECT @Total = COUNT(*) FROM Student;
END;
```

Control Statements In Procedures

Stored procedures support programming constructs.

IF–ELSE Example

```
IF @Marks >= 40
   PRINT 'Pass'
ELSE
   PRINT 'Fail'
```

WHILE Loop Example

```
WHILE @i <= 5
BEGIN
   PRINT @i;
   SET @i = @i + 1;
END;
```

Procedures Vs Functions

Feature	Procedure	Function
Return value	Optional	Mandatory
DML operations	Allowed	Restricted

Feature	Procedure	Function
Called using	EXEC	SELECT
Output parameters	Yes	No

Advantages of Stored Procedures

- Faster execution
- Reduced SQL redundancy
- Improved data integrity
- Better security control
- Easy maintenance

Limitations of Stored Procedures

- Database-dependent syntax
- Difficult debugging
- Increased server load
- Limited portability

Stored Procedures and Security

Stored procedures help improve security by: - Restricting direct table access - Granting EXECUTE permission only

```
GRANT EXECUTE ON GetStudents TO user1;
```

Cursors

Introduction to Cursors

A **Cursor** in DBMS is a database object used to **retrieve, process, and manipulate records row by row** from a result set returned by an SQL query. Normally, SQL works on a set of rows at a time, but cursors allow **tuple-by-tuple processing**.

Definition: > A cursor is a pointer that refers to a result set of a query and allows processing of individual rows.

Why Cursors Are Required

Cursors are needed when: - Row-by-row processing is required - Complex business logic cannot be handled using set-based SQL - Sequential processing of query results is needed - Conditional operations must be applied to each row

Types of Cursors

Implicit Cursors

- Automatically created by DBMS
- Used for INSERT, UPDATE, DELETE
- No explicit declaration required

Example:

```
INSERT INTO Student VALUES (101, 'Ravi', 'CSE', 85);
```

Explicit Cursors

- Created and controlled by the programmer
- Used with SELECT statements
- Require declaration, opening, fetching, and closing

Cursor Life Cycle

An explicit cursor follows these steps: 1. Declare the cursor 2. Open the cursor 3. Fetch rows from the cursor 4. Close the cursor

Syntax of Explicit Cursor

```
DECLARE cursor_name CURSOR FOR
SELECT column1, column2 FROM table_name;

OPEN cursor_name;
FETCH cursor_name INTO variable1, variable2;
CLOSE cursor_name;
```

Example of Cursor Usage

Table: STUDENT

RollNo	Name	Dept	Marks
101	Ravi	CSE	85
102	Anu	ECE	72
103	Kiran	CSE	90

Cursor to Display Student Names

```
DECLARE student_cursor CURSOR FOR
SELECT Name, Marks FROM Student;

OPEN student_cursor;
FETCH student_cursor INTO @Name, @Marks;
WHILE @@FETCH_STATUS = 0
BEGIN
   PRINT @Name + ' ' + CAST(@Marks AS VARCHAR);
   FETCH student_cursor INTO @Name, @Marks;
END;
CLOSE student_cursor;
```

Cursor Attributes

Cursor attributes provide information about cursor execution.

Attribute	Description
%FOUND	TRUE if last fetch returned a row
%NOTFOUND	TRUE if no row returned
%ROWCOUNT	Number of rows fetched

Attribute	Description
%ISOPEN	TRUE if cursor is open

Parameterized Cursors

Cursors can accept parameters to control query execution.

```
DECLARE dept_cursor CURSOR FOR
SELECT Name FROM Student WHERE Dept = @DeptName;
```

Advantages of Cursors

- Allows row-by-row processing
- Useful for complex calculations
- Provides greater control over data manipulation

Disadvantages of Cursors

- Slower than set-based operations
- High memory consumption
- Increased CPU usage
- Not recommended for large datasets

Cursors Vs Set-Based Sql

Feature	Cursor	Set-Based SQL
Processing	Row-by-row	Set-at-a-time
Performance	Slow	Fast
Complexity	High	Low

When to Use Cursors

Use cursors only when: - Set-based SQL is insufficient - Business logic demands row-wise processing

Avoid cursors for simple SELECT, UPDATE, DELETE operations.

Triggers

Introduction to Triggers

A **Trigger** in DBMS is a **special type of stored procedure** that is automatically executed (fired) when a specified event occurs on a table or view. Triggers are mainly used to **maintain data integrity, enforce business rules, and perform automatic actions**.

Definition: > A trigger is a database object that is automatically invoked in response to an INSERT, UPDATE, or DELETE operation on a table or view.

Why Triggers Are Required

Triggers are required to: - Enforce complex business rules - Maintain referential integrity - Automatically validate data - Log database activities (auditing) - Prevent invalid transactions

Events That Activate Triggers

Triggers are activated by the following events: - **INSERT** – when a new record is added - **UPDATE** – when an existing record is modified - **DELETE** – when a record is removed

Types of Triggers

Before Triggers

Executed **before** the triggering operation.

Use case: Validation of data before insertion.

```
CREATE TRIGGER before_insert_student
BEFORE INSERT ON Student
FOR EACH ROW
BEGIN
   IF NEW.Marks < 0 THEN
      SIGNAL SQLSTATE '45000'
      SET MESSAGE_TEXT = 'Marks cannot be negative';
   END IF;
END;
```

After Triggers

Executed **after** the triggering operation.

Use case: Logging or auditing.

```
CREATE TRIGGER after_insert_student
AFTER INSERT ON Student
FOR EACH ROW
BEGIN
   INSERT INTO Student_Log VALUES (NEW.RollNo, CURRENT_DATE);
END;
```

Instead of Triggers

Used mainly on **views** to perform operations instead of the actual action.

```
CREATE TRIGGER instead_of_insert_view
INSTEAD OF INSERT ON Student_View
BEGIN
   INSERT INTO Student VALUES (NEW.RollNo, NEW.Name, NEW.Dept, NEW.Marks);
END;
```

Trigger Timing

Triggers can be classified based on timing: - BEFORE trigger - AFTER trigger - INSTEAD OF trigger

Trigger Syntax (General Form)

```
CREATE TRIGGER trigger_name
BEFORE | AFTER | INSTEAD OF
INSERT | UPDATE | DELETE
ON table_name
FOR EACH ROW
BEGIN
```

```
   SQL statements;
END;
```

Old and New References

Triggers use **OLD** and **NEW** keywords to access data values.

- **OLD.column_name** – previous value (DELETE / UPDATE)
- **NEW.column_name** – new value (INSERT / UPDATE)

Example: Update Trigger

```
CREATE TRIGGER update_salary
BEFORE UPDATE ON Employee
FOR EACH ROW
BEGIN
   IF NEW.Salary < OLD.Salary THEN
      SIGNAL SQLSTATE '45000'
      SET MESSAGE_TEXT = 'Salary cannot be reduced';
   END IF;
END;
```

Advantages of Triggers

- Automatic execution
- Improves data integrity
- Centralized enforcement of rules
- Useful for auditing and logging

Limitations of Triggers

- Difficult to debug
- Hidden execution logic
- Performance overhead
- Complex dependency management

Triggers Vs Stored Procedures

Feature	Trigger	Stored Procedure
Execution	Automatic	Manual
Invocation	Event-driven	Explicit call
Parameters	Not allowed	Allowed

When to Use Triggers

Triggers should be used when: - Automatic rule enforcement is required - Data consistency must be guaranteed - Logging changes is necessary

Avoid using triggers for complex business logic.

Functional Dependencies

Introduction

In a Database Management System (DBMS), **Functional Dependency (FD)** is a constraint that describes the relationship between attributes in a relation. Functional dependencies play a vital role in **database design**, **normalization**, and **maintaining data integrity**.

Definition of Functional Dependency

Let R be a relation schema and X and Y be subsets of attributes of R. A functional dependency **X → Y** holds if, for any two tuples in R, whenever the values of X are the same, the values of Y must also be the same.

 X is called the **determinant**, and Y is the **dependent attribute**.

Example of Functional Dependency

Consider the relation:

STUDENT(USN, Name, DeptID, DeptName)

Functional Dependencies: - USN → Name, DeptID - DeptID → DeptName

This means: - Each USN uniquely identifies a student and their department - Each department ID determines exactly one department name

Types of Functional Dependencies

Trivial Functional Dependency

A functional dependency **X → Y** is said to be trivial if **Y ⊆ X**.

Example: (USN, Name) → Name

Non-Trivial Functional Dependency

A functional dependency is **non-trivial** if **Y ⊄ X**.

Example: USN → Name

Completely Non-Trivial Functional Dependency

A functional dependency is completely non-trivial if **X ∩ Y = ∅**.

Example: USN → DeptID

Partial Dependency

A **partial dependency** occurs when a non-prime attribute depends on **part of a composite key**.

Example: (StudentID, CourseID) → Grade StudentID → StudentName

This violates **Second Normal Form (2NF)**.

Full Functional Dependency

A functional dependency **X** → **Y** is a full functional dependency if Y depends on the **entire X** and not on any proper subset of X.

Example: (StudentID, CourseID) → Grade

Transitive Dependency

A **transitive dependency** exists when: - X → Y - Y → Z

Then X → Z

Example: EmpID → DeptID DeptID → DeptName

This violates **Third Normal Form (3NF)**.

Armstrong's Axioms (Inference Rules)

Armstrong's axioms are a set of rules used to infer all functional dependencies from a given set.

Reflexivity Rule

If Y ⊆ X, then X → Y

Augmentation Rule

If X → Y, then XZ → YZ

Transitivity Rule

If X → Y and Y → Z, then X → Z

Derived Rules

From Armstrong's axioms, the following rules can be derived: - **Union:** If X → Y and X → Z, then X → YZ - **Decomposition:** If X → YZ, then X → Y and X → Z - **Pseudo-transitivity:** If X → Y and YZ → W, then XZ → W

Attribute Closure

The **attribute closure** of a set of attributes X, denoted as $\mathbf{X^+}$, is the set of all attributes functionally determined by X.

Example:

Relation R(A, B, C, D) FDs: - A → B - B → C - C → D

Then: A^+ = {A, B, C, D}

Candidate Key using Functional Dependencies

An attribute set K is a **candidate key** if: - K^+ contains all attributes of the relation - No proper subset of K has this property

Functional dependencies are used to identify candidate keys.

Role of Functional Dependencies in Normalization

- Identify candidate keys
- Detect partial and transitive dependencies
- Basis for normalization into **2NF, 3NF, and BCNF**
- Reduce redundancy and anomalies

Normalization

Introduction

Normalization is a systematic process of organizing data in a database to minimize redundancy and eliminate undesirable characteristics such as insertion, update, and deletion anomalies. It is based on **functional dependencies** and plays a crucial role in designing efficient and consistent relational databases.

Objectives of Normalization

Normalization aims to: - Reduce data redundancy - Eliminate data anomalies - Ensure data integrity and consistency - Simplify database maintenance - Improve database design

Data Anomalies

Insertion Anomaly

Occurs when certain data cannot be inserted without the presence of other data.

Example: Cannot insert a new department if no employee is assigned to it.

Update Anomaly

Occurs when the same data item must be updated in multiple rows, leading to inconsistency.

Example: Updating a department name in multiple tuples.

Deletion Anomaly

Occurs when deleting a record results in loss of important data.

Example: Deleting the last employee of a department removes department information.

Normal Forms

First Normal Form (1NF)

A relation is in **First Normal Form (1NF)** if: - All attributes contain atomic (indivisible) values - There are no repeating groups or multivalued attributes

Example (Not in 1NF): STUDENT(USN, Name, Subjects)

After 1NF: STUDENT(USN, Name, Subject)

4.2 Second Normal Form (2NF)

A relation is in **Second Normal Form (2NF)** if: - It is in 1NF - There is no partial dependency (no non-prime attribute depends on part of a composite key)

Example: ENROLLMENT(StudentID, CourseID, StudentName, CourseName)

Partial Dependencies: - StudentID → StudentName - CourseID → CourseName

Decomposition: - STUDENT(StudentID, StudentName) - COURSE(CourseID, CourseName) - ENROLLMENT(StudentID, CourseID)

Third Normal Form (3NF)

A relation is in **Third Normal Form (3NF)** if: - It is in 2NF - There is no transitive dependency

Example: EMPLOYEE(EmpID, EmpName, DeptID, DeptName)

Dependencies: - EmpID → DeptID - DeptID → DeptName

Decomposition: - EMPLOYEE(EmpID, EmpName, DeptID) - DEPARTMENT(DeptID, DeptName)

Boyce–Codd Normal Form (BCNF)

A relation is in **BCNF** if for every functional dependency X → Y, X is a super key.

Example: COURSE(CourseID, Instructor, Room)

Dependencies: - Instructor → Room - CourseID → Instructor

Instructor is not a super key, so BCNF is violated.

Fourth Normal Form (4NF)

A relation is in **Fourth Normal Form (4NF)** if: - It is in BCNF - It has no multivalued dependencies

Example: A student having multiple skills and multiple hobbies independently.

Fifth Normal Form (5NF)

A relation is in **Fifth Normal Form (5NF)** if: - It is decomposed based on join dependencies - Further decomposition results in loss of information

Used in rare and complex database designs.

Normalization vs Denormalization

Aspect	Normalization	Denormalization
Redundancy	Low	High
Consistency	High	Lower
Query Performance	Slower	Faster
Maintenance	Easy	Difficult

Advantages of Normalization

- Eliminates redundancy
- Prevents anomalies
- Improves data consistency
- Enhances database design

Disadvantages of Normalization

- Increased number of tables

- Complex queries due to joins
- May affect performance

Properties of Good Decomposition

Introduction

In Database Management Systems (DBMS), **decomposition** is the process of breaking a large relation (table) into smaller relations to reduce redundancy and avoid anomalies. However, decomposition must be done carefully. A **good decomposition** satisfies certain properties to ensure that the original information is not lost and database constraints are preserved.

The two most important properties of good decomposition are: 1. **Lossless-Join Decomposition** 2. **Dependency Preservation**

Why Decomposition is Needed

Decomposition is mainly required to: - Eliminate data redundancy - Avoid update, insertion, and deletion anomalies - Improve data integrity - Achieve higher normal forms (2NF, 3NF, BCNF)

Properties of Good Decomposition

Lossless-Join Decomposition

Definition

A decomposition is said to be **lossless-join** if the original relation can be **reconstructed exactly** by performing a natural join on the decomposed relations, without generating spurious tuples.

Condition for Lossless-Join

Let a relation R be decomposed into R1 and R2. The decomposition is lossless if **either** of the following is true: - $R1 \cap R2 \rightarrow R1$ - $R1 \cap R2 \rightarrow R2$

(Where $\rightarrow$ denotes functional dependency)

Example of Lossless Decomposition

Relation: R(StudentID, Name, DeptID)

Functional Dependency: StudentID $\rightarrow$ Name, DeptID

Decomposition: - R1(StudentID, Name) - R2(StudentID, DeptID)

Here, StudentID is a common attribute and is a key for both relations. Joining R1 and R2 on StudentID will recreate the original relation without loss.

Lossy Decomposition (Bad Decomposition)

If joining the decomposed tables produces **extra tuples**, the decomposition is **lossy**.

Example: R(A, B, C) $\rightarrow$ R1(A, B) and R2(B, C) If B is not a key, spurious tuples may appear after join.

Dependency Preservation

Definition

A decomposition is said to be **dependency preserving** if all **functional dependencies** of the original relation can be enforced by enforcing dependencies on the individual decomposed relations **without performing a join**.

Importance of Dependency Preservation

- Avoids expensive joins for constraint checking
- Ensures data integrity efficiently
- Simplifies database maintenance

Example of Dependency Preservation

Relation: R(A, B, C)

Functional Dependencies: A → B B → C

Decomposition: - R1(A, B) - R2(B, C)

Both dependencies can be enforced locally in R1 and R2, hence the decomposition is dependency preserving.

Non-Preserving Decomposition

If a dependency can be enforced only by joining tables, then the decomposition is **not dependency preserving**.

Trade-off Between the Two Properties

Sometimes, it is not possible to achieve both properties together: - **BCNF** ensures lossless-join but may lose dependency preservation - **3NF** allows dependency preservation with lossless-join

Hence, 3NF is often preferred in practical database design.

Summary Table

Property	Meaning	Importance
Lossless-Join	No information loss	Data correctness
Dependency Preservation	All dependencies preserved	Data integrity

Joins

Introduction

In a relational database, data is stored in multiple tables to avoid redundancy through normalization. To retrieve meaningful information from these related tables, **joins** are used. A join combines rows from two or more tables based on a related column, typically a **primary key–foreign key relationship**.

Need for Joins

Joins are required to: - Retrieve data spread across multiple tables - Maintain data consistency in normalized databases - Establish relationships between entities - Avoid data redundancy

Types of Joins in DBMS

Inner Join

An **INNER JOIN** returns only those records that have matching values in both tables.

Syntax:

```
SELECT columns
FROM Table1
INNER JOIN Table2
ON Table1.common_column = Table2.common_column;
```

Example:

```
SELECT E.EmpName, D.DeptName
FROM Employee E
INNER JOIN Department D
ON E.DeptID = D.DeptID;
```

Explanation: Only employees who belong to a department are displayed.

Left Outer Join (LEFT JOIN)

A **LEFT JOIN** returns all records from the left table and matching records from the right table. If no match exists, NULL values are returned for the right table.

Syntax:

```
SELECT columns
FROM Table1
LEFT JOIN Table2
ON condition;
```

Example:

```
SELECT E.EmpName, D.DeptName
FROM Employee E
LEFT JOIN Department D
ON E.DeptID = D.DeptID;
```

Right Outer Join (RIGHT JOIN)

A **RIGHT JOIN** returns all records from the right table and matching records from the left table.

Example:

```
SELECT E.EmpName, D.DeptName
FROM Employee E
RIGHT JOIN Department D
ON E.DeptID = D.DeptID;
```

Full Outer Join (FULL JOIN)

A **FULL OUTER JOIN** returns all records from both tables. Unmatched rows contain NULL values.

Example:

```
SELECT E.EmpName, D.DeptName
FROM Employee E
FULL OUTER JOIN Department D
ON E.DeptID = D.DeptID;
```

Cross Join

A **CROSS JOIN** returns the Cartesian product of both tables.

Example:

```
SELECT E.EmpName, D.DeptName
FROM Employee E
CROSS JOIN Department D;
```

Note: If Employee has m rows and Department has n rows, the result will have m × n rows.

Self Join

A **SELF JOIN** is used to join a table with itself. Table aliases are required.

Example:

```
SELECT E.EmpName AS Employee, M.EmpName AS Manager
FROM Employee E
LEFT JOIN Employee M
ON E.ManagerID = M.EmpID;
```

Natural Join

A **NATURAL JOIN** automatically joins tables based on columns with the same name and data type.

Example:

```
SELECT *
FROM Employee
NATURAL JOIN Department;
```

Limitation: Not recommended in practice due to ambiguity.

Equi Join

An **EQUI JOIN** uses the equality operator (=) in the join condition.

Example:

```
SELECT *
FROM Employee E, Department D
WHERE E.DeptID = D.DeptID;
```

Non-Equi Join

A **NON-EQUI JOIN** uses operators other than '=' such as <, >, BETWEEN.

Example:

```
SELECT E.EmpName, S.Grade
FROM Employee E
JOIN SalaryGrade S
ON E.Salary BETWEEN S.MinSal AND S.MaxSal;
```

Comparison of Join Types

Join Type	Description
Inner Join	Returns matching rows only
Left Join	All rows from left table
Right Join	All rows from right table
Full Join	All rows from both tables
Cross Join	Cartesian product
Self Join	Table joined with itself

Joins vs Nested Queries

Aspect	Joins	Nested Queries
Performance	Faster	Slower
Readability	Moderate	High
Optimization	Easy	Difficult
Use Case	Large datasets	Conditional filtering

Transactions, Locking, and Serializability

Introduction

In a Database Management System (DBMS), multiple users can access and modify data simultaneously. To ensure **data consistency, integrity, and reliability**, DBMS uses **transactions, locking mechanisms, and serializability concepts**. This document explains these concepts with examples for better understanding.

Transactions in DBMS

Definition

A **transaction** is a sequence of database operations that forms a single logical unit of work. Transactions must satisfy **ACID properties**.

Example: Bank money transfer: 1. Debit ₹10,000 from Account A 2. Credit ₹10,000 to Account B

If any step fails, the entire transaction is rolled back.

Transaction Operations

Operation	Description
READ(X)	Reads the value of data item X
WRITE(X)	Writes a new value to data item X

Operation	Description
COMMIT	Makes all changes permanent
ROLLBACK	Undoes all changes of the transaction

SQL Example:

```
START TRANSACTION;
UPDATE Account SET balance = balance - 10000 WHERE acc_no = 101;
UPDATE Account SET balance = balance + 10000 WHERE acc_no = 102;
COMMIT;
```

Concurrency Problems with Examples

Lost Update

Two transactions update the same data simultaneously.

Example: - T1 reads balance = 50,000 - T2 reads balance = 50,000 - T1 updates to 60,000 - T2 updates to 55,000 (overwrites T1)

Dirty Read

A transaction reads uncommitted changes from another transaction.

Example: - T1 updates balance to 60,000 (not committed) - T2 reads 60,000 - T1 rolls back, balance is 50,000, T2 has wrong data

Non-repeatable Read

Same transaction reads the same row twice but gets different values.

Example: - T1 reads salary = 50,000 - T2 updates salary to 55,000 and commits - T1 reads again, salary = 55,000

Phantom Read

New rows appear in a query result when re-executed.

Example: - T1: SELECT * FROM Employee WHERE Dept='IT' - T2 inserts a new IT employee and commits - T1 executes the SELECT again and sees new row

Locking Protocols

Introduction

In a Database Management System (DBMS), **locking protocols** define the rules for how transactions acquire and release locks on data items. These protocols ensure **data consistency, isolation, and serializability** when multiple transactions execute concurrently.

Why Locking Protocols are Needed

Without locking protocols, concurrent transactions may lead to: - Lost updates - Dirty reads - Non-repeatable reads - Phantom reads

Locking protocols coordinate transaction execution to avoid these problems.

Basic Lock Types

Lock Type	Description	Example
Shared Lock (S)	Used for READ operation	Multiple transactions read balance
Exclusive Lock (X)	Used for WRITE operation	Only one transaction updates balance

Compatibility Matrix

Introduction

In a Database Management System (DBMS), when multiple transactions access the same data item concurrently, **locks** are used to control access. The **Compatibility Matrix** defines whether two different types of locks can be held on the same data item at the same time without causing conflicts.

The compatibility matrix is a crucial concept in **concurrency control and locking protocols**.

What is Lock Compatibility?

Lock compatibility determines: - Whether a lock request can be **granted immediately** - Or whether the requesting transaction must **wait**

If two locks are compatible, they can coexist. If they are incompatible, one transaction must wait.

Types of Locks Considered

Lock Type	Description
Shared Lock (S)	Used for READ operations
Exclusive Lock (X)	Used for WRITE operations

Compatibility Matrix

The **lock compatibility matrix** shows which lock types are compatible with each other.

Requested Held	Shared (S)	Exclusive (X)
Shared (S)	✓ Compatible	✕ Not Compatible
Exclusive (X)	✕ Not Compatible	✕ Not Compatible

Explanation of Compatibility Matrix

Shared vs Shared (S–S)

- Multiple transactions can read the same data simultaneously
- No data inconsistency
- ✓ Compatible

Example: - T1 reads Account balance - T2 reads Account balance at the same time

Shared vs Exclusive (S–X)

- One transaction wants to read, another wants to write
- Writing may change data being read
- ✕ Not Compatible

Example: - T1 reads salary - T2 tries to update salary → must wait

Exclusive vs Shared (X–S)

- One transaction is writing
- Other transaction cannot read uncommitted data
- ✕ Not Compatible

Example: - T1 updates marks - T2 wants to read marks → must wait

Exclusive vs Exclusive (X–X)

- Two transactions cannot write simultaneously
- Causes lost updates
- ✕ Not Compatible

Example: - T1 updates balance - T2 updates balance at the same time → conflict

Role of Compatibility Matrix in DBMS

- Used by **lock manager** to decide lock granting
- Prevents concurrency anomalies
- Ensures isolation and serializability
- Fundamental for **Two-Phase Locking (2PL)**

Compatibility Matrix with Example Schedule

Time	T1	T2
t1	S-lock(A)	
t2	READ(A)	S-lock(A)
t3		READ(A)
t4		X-lock(A) ✕(wait)
t5	UNLOCK(A)	
t6		X-lock(A) ✓

Types of Locking Protocols with Examples

Binary Locking Protocol

Definition

In **binary locking**, a data item is either **locked (1)** or **unlocked (0)**.

Rules

- A transaction must lock the data item before accessing it

- Unlock after the operation

Example

- T1 locks data item A
- T1 reads/writes A
- T1 unlocks A
- T2 can now access A

Disadvantages

- Low concurrency
- Deadlocks may occur

Multiple Locking Protocol

Definition

Allows different lock modes such as **Shared (S)** and **Exclusive (X)**.

Example

- T1 acquires S-lock on Account A to read balance
- T2 also acquires S-lock on Account A
- T3 requests X-lock → must wait until T1 and T2 release S-locks

Advantage

- Better concurrency than binary locking

Two-Phase Locking Protocol (2PL)

Definition

A transaction follows two phases: 1. **Growing Phase** – Only lock acquisition 2. **Shrinking Phase** – Only lock release

Example

Transaction T1: - Lock(A) - Lock(B) - Read(A), Write(B) - Unlock(A) - Unlock(B)

Once T1 releases a lock, it cannot acquire any new lock.

Property

- Guarantees **conflict serializability**

Disadvantage

- Deadlock possible

Strict Two-Phase Locking (Strict 2PL)

Definition

All **exclusive locks** are held until commit or abort.

Example

- T1 acquires X-lock on A

- Updates A
- Commits transaction
- Releases X-lock on A

Other transactions cannot read or write A until T1 commits.

Advantage

- Prevents dirty reads
- Ensures recoverability

Rigorous Two-Phase Locking

Definition

Both **shared and exclusive locks** are released only after commit or abort.

Example

- T1 acquires S-lock on A and X-lock on B
- Performs operations
- Commits
- Releases all locks together

Advantage

- Simplifies recovery

Disadvantage

- Very low concurrency

Conservative Two-Phase Locking (Static 2PL)

Definition

A transaction acquires **all required locks before it begins execution**.

Example

- T1 requests locks on A, B, and C
- If all locks are granted, T1 starts execution
- Otherwise, T1 waits

Advantage

- Deadlock-free

Disadvantage

- Low concurrency

Comparison of Locking Protocols

Protocol	Serializability	Deadlock	Concurrency
Binary Locking	No	Yes	Low
Multiple Locking	No	Yes	Medium
Two-Phase Locking	Yes	Yes	Medium

Protocol	Serializability	Deadlock	Concurrency
Strict 2PL	Yes	Yes	Low
Rigorous 2PL	Yes	Yes	Very Low
Conservative 2PL	Yes	No	Low

Serializability

Introduction

In a Database Management System (DBMS), multiple transactions are often executed **concurrently** to improve system performance. However, concurrent execution may lead to data inconsistency. **Serializability** is a concept used to ensure that concurrent transaction execution produces results equivalent to some **serial execution** of those transactions.

What is a Schedule?

A **schedule** is the sequence in which operations of multiple transactions are executed.

Types of Schedules

- **Serial Schedule**: Transactions execute one after another without interleaving.
- **Non-Serial Schedule**: Operations of transactions are interleaved.

Definition of Serializability

A schedule is said to be **serializable** if its result is the same as the result of some **serial schedule**, even though transactions execute concurrently.

Serializability ensures **correctness and consistency** of the database.

Need for Serializability

Serializability is required to: - Maintain database consistency - Avoid concurrency anomalies - Ensure isolation property of ACID - Allow safe concurrent execution of transactions

Types of Serializability

Conflict Serializability

Definition

A schedule is **conflict serializable** if it can be transformed into a serial schedule by swapping **non-conflicting operations**.

Conflicting Operations

Two operations conflict if: - They belong to different transactions - They operate on the same data item - At least one of them is a WRITE operation

Types of conflicts: - Read–Write (RW) - Write–Read (WR) - Write–Write (WW)

Precedence Graph (Serialization Graph)

- Nodes represent transactions
- Directed edge Ti → Tj indicates a conflict where Ti precedes Tj
- If the graph has **no cycle**, the schedule is conflict serializable

Example of Conflict Serializability

Transactions: - T1: READ(A), WRITE(A) - T2: READ(A), WRITE(A)

If the precedence graph has no cycle, the schedule is conflict serializable.

View Serializability

Definition

A schedule is **view serializable** if it is **view equivalent** to a serial schedule.

Conditions for View Equivalence

Two schedules are view equivalent if: 1. Initial reads are the same 2. Reads-from relationships are preserved 3. Final writes are on the same data items

Example of View Serializability

A schedule may be view serializable even if it is **not conflict serializable**.

Recovery Methods

Introduction

In a Database Management System (DBMS), failures such as system crashes, power failures, or transaction errors may occur during execution. **Recovery methods** are techniques used to restore the database to a **consistent state** after a failure. Recovery ensures the **Atomicity and Durability** properties of ACID.

Types of Failures in DBMS

Transaction Failure

Occurs due to logical errors, constraint violations, or deadlocks.

System Failure

Occurs due to power failure, OS crash, or hardware issues. Main memory contents are lost.

Disk Failure

Occurs when secondary storage crashes, leading to loss of database data.

Recovery Concepts

Stable Storage

A storage medium that survives system failures (e.g., disks, backups).

Log-Based Recovery

DBMS maintains a **log file** that records all changes made by transactions.

Log-Based Recovery Methods

Transaction Log

A log contains records such as: - <T, START> - <T, WRITE, X, old value, new value> - <T, COMMIT> - <T, ABORT>

Deferred Database Modification

Concept

- Database updates are applied **only after commit**
- Log is written first

Recovery Rule

- REDO committed transactions
- No UNDO needed

Example

If T1 commits before crash, REDO T1 during recovery.

Immediate Database Modification

Concept

- Database is updated immediately
- Changes may exist before commit

Recovery Rule

- UNDO uncommitted transactions
- REDO committed transactions

Example

If T1 committed and T2 not committed: - REDO T1 - UNDO T2

Checkpoint-Based Recovery

Definition

A **checkpoint** is a point where DBMS saves the current state of the database and log.

Advantages

- Reduces recovery time
- Limits log scanning

Recovery Steps

1. Start from last checkpoint
2. REDO committed transactions
3. UNDO uncommitted transactions

Shadow Paging

Concept

- Maintains two page tables: shadow and current
- Updates are made on current pages

Recovery

- If crash occurs, revert to shadow page table

Advantages

- No logging required

Disadvantages

- Poor performance
- Not suitable for large databases

Backup-Based Recovery

Concept

- Periodic backups of the database
- Used in case of disk failure

Steps

1. Restore last backup
2. Apply logs after backup

Comparison of Recovery Methods

Method	Logging Required	UNDO	REDO
Deferred Update	Yes	No	Yes
Immediate Update	Yes	Yes	Yes
Checkpoint	Yes	Yes	Yes
Shadow Paging	No	No	No

Relation with ACID Properties

- **Atomicity**: Ensured by UNDO
- **Durability**: Ensured by REDO and backups

Questions

1. What is the difference between a super key and a candidate key?
2. Why is a primary key not allowed to contain NULL values?
3. Can a relation have more than one candidate key? Explain with an example.
4. What is a composite key and where is it used?
5. Differentiate between primary key and foreign key.
6. How does a unique key differ from a primary key?
7. Why are keys important for maintaining data integrity?
8. Define functional dependency with an example.
9. What is a trivial functional dependency?
10. Explain partial dependency and name the normal form it violates.
11. What is transitive dependency? Give an example.
12. What is attribute closure and why is it used?
13. Explain Armstrong's axioms.
14. How do functional dependencies help in identifying candidate keys?
15. What is normalization and why is it required?
16. Explain insertion, deletion, and update anomalies.
17. What are the conditions for a table to be in First Normal Form (1NF)?
18. How does Second Normal Form (2NF) eliminate partial dependency?
19. Explain Third Normal Form (3NF).
20. Why is BCNF considered stronger than 3NF?
21. Can a relation be in 3NF but not in BCNF? Why?
22. What is decomposition in DBMS?
23. Define lossless-join decomposition.
24. What condition ensures a decomposition is lossless?
25. What is dependency preservation?
26. Why is lossless-join property mandatory?
27. Why is dependency preservation sometimes sacrificed in BCNF?
28. Why is 3NF often preferred over BCNF in practice?
29. What is a join and why is it required?
30. Explain INNER JOIN with an example.
31. Differentiate between LEFT JOIN and RIGHT JOIN.
32. What is a FULL OUTER JOIN?
33. What is a SELF JOIN?
34. Explain CROSS JOIN and its output.
35. Difference between EQUI JOIN and NON-EQUI JOIN.
36. What are aggregate operators in SQL?
37. Explain COUNT(), SUM(), and AVG() functions.
38. What is the role of GROUP BY clause?
39. Difference between WHERE and HAVING clauses.
40. How are NULL values handled in aggregate functions?
41. Can aggregate functions be used without GROUP BY? Explain.
42. What is a nested query?
43. Difference between single-row and multi-row subqueries.

44. What is a correlated query?
45. Why are correlated queries slower than normal nested queries?
46. Difference between nested queries and joins.
47. When should nested queries be preferred over joins?
48. What is a transaction in DBMS?
49. Explain the Atomicity property with an example.
50. How does Consistency differ from Atomicity?
51. What problems occur without Isolation?
52. How is Durability ensured in DBMS?
53. What is the role of COMMIT and ROLLBACK?
54. What is Data Control Language (DCL)?
55. Explain GRANT and REVOKE commands.
56. What is Transaction Control Language (TCL)?
57. Explain COMMIT, ROLLBACK, and SAVEPOINT.
58. How does SAVEPOINT help in transaction management?
59. Differentiate between DCL and TCL.
60. What is a schedule in DBMS?
61. Difference between serial and non-serial schedules.
62. Why is serializability important?
63. What are shared and exclusive locks?
64. Explain the lock compatibility matrix.
65. What is Two-Phase Locking (2PL)?
66. Why does Strict 2PL improve recoverability?
67. What is recovery in DBMS?
68. What information is stored in a transaction log?
69. Difference between deferred update and immediate update recovery.
70. What is a checkpoint and why is it used?
71. Why is Write-Ahead Logging (WAL) important?
72. What is shadow paging and why is it not commonly used?