1. *Active State:*

- *Description:* The transaction is in progress. Operations like reading and writing data are beingperformed.

  - *Read:* Fetch data from the database.

  - *Write:* Update or insert data into the database.

  - *Savepoint:* A point within a transaction towhich a rollback can be performed.

### 2. *Partially Committed State:*

- *Description:* The transaction has completed its execution but is waiting for the final commit operation.

  - *Commit:* The transaction's changes are made permanent in the database.

  - *Rollback to Savepoint:* Revert the transaction to a specific savepoint without affecting the rest of the transaction.

### 3. *Committed State:*

The transaction has been successfully completed, and all changes are now permanent in the database.

  - *None:* The transaction is complete, and no further operations are allowed on this transaction.

### 4. *Failed State:*

- *Description:* The transaction has encountered an error and cannot proceed.

  - ***Rollback:*** Revert all changes made duringthe transaction to bring the database back to its initial state before the transaction began.

### 5. *Aborted State:*

- *Description:* The transaction has been rolled back due to a failure or a manual intervention. The

database is restored to the state it was in before the transaction began.

- *Operations:*

  - *Retry:* The transaction might be retried after resolving the cause of the failure.

  - *Abort:* The transaction is permanently aborted, and no retry is performed.

### 6. *Terminated State:*

- *Description:* The transaction has reached its final state, whether committed or aborted.

  - *None:* The transaction has completed its lifecycle and cannot be altered.

MODULE 3

Update anomalies in SQL occur when changes to the database result in unintended side effects, often due to poor database design, particularly in a non-normalized database. These anomalies can be categorized into three types: *Insertion Anomaly, **Update Anomaly, and **Deletion Anomaly*. Here's an explanation of each with examples:

### 1. *Insertion Anomaly*

- *Definition*: An insertion anoma

- ly occurs when certain data cannot be inserted into the database without the presence of other data. Thistypically happens in databases that are not fully normalized, where a single table may store multiple types of information.

Example: Consider a table EmployeeDepartment where each row contains EmployeeID, EmployeeName, DepartmentID,and DepartmentName. If you want to add a new department that currently has no employees, you may be forced to insert a dummy employee to ensure the department is recorded in the table.

plaintext

EmployeeDepartment

Table:

EmployeeID	EmployeeName	DepartmentID	DepartmentName
1	John Doe	101	HR
2	Jane Smith	102	IT

Issue: You cannot insert a new department (e.g., `Marketing`) unless there is at least one employeeassociated with it.

### 2. *Update Anomaly*

- *Definition*: An update anomaly occurs when changes to the data in one row require changes to the same data in multiple rows. If these changes are not made consistently, it leads to inconsistentdata.

- *Example*: Using the same EmployeeDepartment table, suppose the name of the HR department is changed to Human Resources. If this change needs to be made, it has to be updated for every employee who works in the HR department.

plaintext

Before Update:

EmployeeID	EmployeeName	DepartmentID	DepartmentName
1	John Doe	101	HR
2	Alice Jones	101	HR
3	Bob Brown	102	IT

After Update:

EmployeeID	EmployeeName	DepartmentID	DepartmentName
1	John Doe	101	Human Resources
2	Alice Jones	101	Human Resources
3	Bob Brown	102	IT

Issue: If the department name is not updated in all rows, some rows might still show `HR`, leading to inconsistent data.

### 3. *Deletion Anomaly*

- *Definition*: A deletion anomaly occurs when the deletion of data representing one fact causes unintended loss of additional data.

MODULE:1

1. ER ATTRIBUTES

4. Single-Valued Attributes

- **Definition**: Attributes that hold only a single value for each entity instance.
- **Example**:
  - For an entity Student, the attribute RollNumber would be single-valued because each student has one unique roll number.

---

5. Multi-Valued Attributes

- **Definition**: Attributes that can hold multiple values for each entity instance.
- **Example**:
  - For an entity Person, the attribute PhoneNumbers may hold multiple values (e.g., home, work, and mobile numbers).
- In ER diagrams, multi-valued attributes are typically represented by double ovals.

---

7. Optional (Nullable) Attributes

- **Definition**: Attributes that may or may not have a value for every instance of an entity.
- **Example**:
  - An attribute MiddleName may be optional, as not all individuals have a middle name.

---

8. Complex Attributes

- **Definition**: Attributes that combine multi-valued and composite characteristics.
- **Example**:
  - An attribute EducationalQualifications might have multiple entries, and each entry could be a composite attribute (e.g., Degree, Institution, Year).

---

9. Stored Attributes

- **Definition**: Attributes that are physically stored in the database and not derived.
- **Example**:
  - EmployeeName is stored directly in the database.

2;RELATIONAL OPERATIONS;

JOIN Operation

- Purpose: Combines tuples from two relations based on a related attribute.
- Symbol: ⋈
  -
- Example: If R1 has attributes (A, B) and R2 has attributes (B, C):
-

2. DIFFERENCE Operation

- Purpose: Finds tuples that are present in one relation but not in another.
- Symbol: −
- Example: If R1 has tuples {(1, A), (2, B), (3, C)} and R2 has tuples {(2, B), (3, C)}:

3. SELECT Operation

- Purpose: Retrieves tuples from a relation that satisfy a given predicate.
- Symbol: σ (Sigma)
- Syntax:

scss

Copy code

σ_condition(Relation)

- Example: If Employee has attributes (EmpID, Name, Age) and rows {(1, John, 25), (2, Jane, 30)}, the query**:**

4. UNION Operation

- Purpose: Combines tuples from two relations into a single relation, eliminating duplicates.
- Symbol: ∪
- Description: The two relations must have the same schema for the union operation to be valid.
- Example: If R1 has tuples {(1, A), (2, B)} and R2 has tuples {(2, B), (3, C)}:

1. UNION Operation

- Purpose: Combines the tuples from two relations into a single relation, eliminating duplicates.
- Symbol: ∪
- Requirements:
    - Both relations must have the same schema (same attributes with the same domain).
- Example: If R1 has tuples {(1, A), (2, B)} and R2 has tuples {(2, B), (3, C)}:

---

2. INTERSECTION Operation

- Purpose: Retrieves the tuples that are present in both relations.
- Symbol: ∩
- Requirements:
    - Both relations must have the same schema (same attributes with the same domain).
- Example: If R1 has tuples {(1, A), (2, B)} and R2 has tuples {(2, B), (3, C)}:

---

3. MINUS (DIFFERENCE) Operation

- Purpose: Retrieves tuples that are in the first relation but not in the second.
- Symbol: −
- Requirements:
    - Both relations must have the same schema (same attributes with the same domain).
- Example: If R1 has tuples {(1, A), (2, B)} and R2 has tuples {(2, B), (3, C)}:
- Use Case: Find records that are unique to one relation.

Key Points

1. **Schema Compatibility: All set theory operations require the relations to have the same schema (arity and domain of attributes must match).**
2. **Duplicates: Duplicates are automatically removed unless otherwise specified in implementations (SQL UNION ALL includes duplicates).**

NoSQL

NoSQL refers to a category of non-relational databases designed to handle large-scale, flexible, and distributed data storage. Unlike traditional relational databases, NoSQL databases are optimized for specific use cases such as scalability, high performance, and schema-less data structures.

Characteristics of NoSQL Databases

1. Schema Flexibility: NoSQL databases do not require a fixed schema, allowing for dynamic and unstructured data storage.
2. Horizontal Scalability: They scale by adding more servers to handle increased load (sharding).
3. High Performance: Optimized for read/write operations and distributed systems.
4. Types of Data: Can store semi-structured, unstructured, and structured data.
5. Variety of Models: NoSQL databases use different data models suited to specific needs.

CAP Theorem

The CAP theorem (also known as Brewer's theorem) states that in a distributed database system, it is impossible to achieve all three of the following properties simultaneously. Systems can only guarantee two at a time:

1. Consistency (C):
    - All nodes in the system see the same data at the same time. A read returns the most recent write.
    - Example: Traditional relational databases prioritize consistency.
2. Availability (A):
    - The system is always operational, and every request gets a response, even if some nodes are down.
    - Example: DNS systems prioritize availability.
3. Partition Tolerance (P):
    - The system continues to function even if there is a network partition or communication breakdown between nodes.
    - Example: Distributed NoSQL

databases prioritize partition tolerance.

Trade-offs in CAP

- Consistency + Availability (CA):
    - Works well in systems without network partitions. Rare in distributed systems because networks are prone to partitions.
    - Example: Relational databases in a single-node setup.
- Consistency + Partition Tolerance (CP):
    - Ensures data consistency even during network partitions but sacrifices availability.
    - Example: MongoDB (with strict consistency settings).
- Availability + Partition Tolerance (AP):
    - Ensures availability and tolerance for network partitions but sacrifices strict consistency.
    - Example: DynamoDB, Cassandra.

The *Two-Phase Locking (2PL) protocol*

is awidely used method for concurrency control in database systems. It ensures serializability, which is the highest level of isolation between transactions, by managing access to shared resources using locks. The protocol is designed to prevent conflicts between transactions that access and modify the same data simultaneously.

### *Overview of Two-Phase Locking (2PL)*

The Two-Phase Locking protocol operates in two distinct phases:

1. *Growing Phase*: In this phase, a transaction can acquire (but not release) any number of locks. The transaction collects all the locks it needs to access and modify data items.

2. *Shrinking Phase*: Once a transaction releases a lock, it cannot acquire any new locks. During this phase, the transaction only releases locks but does not request any additional locks.

#### *1. Growing Phase:*

- ***Lock Acquisition***: During this phase, the transaction requests and obtains locks on the data items it needs to access or modify. The types of locks include:

- ***Exclusive Lock (X-lock)*:** Allows both read and write access to the data item. No other transaction can acquire any lock on this data item while it is held.

- ***Shared Lock (S-lock)*:** Allows read-only access to the data item. Multiple transactions can hold shared locks on the same data item simultaneously, but no transaction can acquire an exclusive lock on it while shared locks are held.

- ***No Lock Release*:** The transaction can continue to acquire additional locks as needed, but cannot release any locks during this phase.

#### *2. Shrinking Phase:*

- ***Lock Release*:** Once the transaction releases a lock, it enters the shrinking phase. During this phase, the transaction can only release locks and cannot acquire new ones.

Commit or Abort: The transaction either commits, making all its changes permanent, or aborts, rolling back any changes made during the transaction.

Example of Two-Phase Locking*

Consider two transactions, T1 and T2, operating on a shared database with a single data item, X.

Transaction T1 needs to read and write data item X.

- *Transaction T2* needs to read data item X.

Sequence of Operations:

1. *Transaction T1* starts and requests an exclusive lock (X-lock) on X.

2. *Transaction T1* acquires the X-lock on X

3. *Transaction T1* then. releases the X-lock

### 1. *Assertions:*

Assertions are conditions or constraints that must hold true for the data in a database. They are typically used to enforce rules that apply to multiple tables or that cannot be easily enforced by standard constraints (like CHECK, UNIQUE, or FOREIGN KEY).

#### Example of Assertion:

Suppose we have a database for a university, with tables for Students and Courses. We want to ensure that no student is allowed to enroll in more than 5 courses. This is a rule that spans across the Enrollments table, which records which students are enrolled in which courses.

Here is how you could create an assertion in SQLto enforce this rule:

sql

```
CREATE ASSERTION max_courses_per_student

CHECK (

   NOT EXISTS (

      SELECT student_id

      FROM Enrollments

      GROUP BY student_id

      HAVING COUNT(course_id) > 5

   )

);
```

This assertion checks that there is no student (student_id) in the Enrollments table who is enrolled in more than 5 courses. If an attempt is made to violate this condition, the database will reject the operation.

Note: Assertions are not supported in many database systems like MySQL or PostgreSQL. They are more theoretical in nature and are used in academic discussions. In practice, triggers are often used to implement similar logic.

Triggers are database objects that automatically execute or "fire" when certain events occur on a table, such as INSERT, UPDATE, or DELETE. Triggers can be used to enforce complex business rules, maintain audit trails, or synchronize tables.

#### Example of Trigger:

Let’s say we have an Accounts table for a bank, and we want to ensure that no account can have a balance that goes below zero. If an UPDATE or INSERT operation tries to reduce the balance to a negative value, the trigger will prevent the transaction.

Here is an example trigger in SQL:

sql

```
CREATE TRIGGER check_balance

BEFORE INSERT OR UPDATE ON Accounts

FOR EACH ROW

BEGIN

   IF NEW.balance < 0 THEN

      SIGNAL SQLSTATE '45000'

      SET MESSAGE_TEXT = 'Balance cannotbe negative';

   END IF;

END;
```

- *BEFORE INSERT OR UPDATE*: This trigger will fire before an INSERT or UPDATE operation on the Accounts table.

- *FOR EACH ROW*: This indicates that the trigger checks each row that is inserted or updated.

- *IF NEW.balance < 0 THEN*: This condition checks if the new balance is less than zero.

- *SIGNAL SQLSTATE '45000'*: This statement throws an error and prevents the operation from proceeding if the condition is true.

NoSQL Graph Database

A **Graph Database** is a type of **NoSQL database** designed to represent and query data structured as graphs. Data is stored as **nodes**, **edges**, and **properties**, making it ideal for applications that require understanding and traversing relationships between data elements.

Core Components of a Graph Database

1. **Nodes**:
   - Represent entities or objects in the graph.
   - Examples: People, places, products.
2. **Edges**:
   - Represent relationships between nodes.
   - Examples: "FRIEND_OF", "PURCHASED", "LOCATED_IN".
   - Edges can have directionality (directed or undirected).
3. **Properties**:
   - Attributes of nodes or edges.
   - Examples: A Person node may have properties like name and age. An FRIEND_OF edge may have a since property.
4. **Labels**:
   - Tags assigned to nodes to define their type or category.
   - Example: A node labeled Person or Product.

Key Features of Graph Databases

1. **Relationship-Oriented**:
   - Relationships are first-class citizens, enabling fast traversal and queries on complex connections.
2. **Schema-Free**:
   - Graph databases are flexible, allowing dynamic addition of nodes, edges, and properties without a predefined schema.
3. **Efficient Traversal**:
   - Optimized for queries involving relationships, such as finding shortest paths, neighbors, or patterns.
4. **ACID Compliance**:
   - Many graph databases are ACID-compliant, ensuring data consistency for transactional applications.

The *ACID properties* are a set of four key principles that ensure reliable and secure transactions in a database system. These properties are crucial for maintaining data integrity and consistency, even in the event of system failures or concurrent access by multiple users. ACID stands for *Atomicity, **Consistency, **Isolation, and **Durability*.

### *1. Atomicity*

Definition: Atomicity ensures that a transaction is treated as a single, indivisible unit of work. It guarantees that either all operations within a transaction are completed successfully, or none of them are applied. If a transaction encounters an error or failure at any point, the entire transaction is rolled back to its initial state.

Example:

Consider a bank transfer where $100 is moved from Account A to Account B. The transaction involves two operations:

1. Subtract $100 from Account A.

2. Add $100 to Account B.

If an error occurs after subtracting from Account A but before adding to Account B, atomicity ensures that the system will roll back the subtraction, so Account A is not left with $100 less while Account B remains unchanged.

### *2. Consistency*

Definition: Consistency ensures that a transaction brings the database from one valid state to another valid state, preserving the database's integrity constraints. It ensures that all rules, constraints, and data integrity conditions are maintained before and after the transaction.

Example:

If a transaction involves updating a customer's address, consistency ensures that the new address adheres to all predefined rules (e.g., correct postal code format). If the address update violates any integrity constraints, the transaction is rolled back, maintaining the database in a consistent state.

### *3. Isolation*

Definition: Isolation ensures that the operations of one transaction are not visible to other transactions until the transaction is committed. It prevents transactions from interfering with each other, which could lead to inconsistent data. Isolation levels vary, ranging from strict isolation (where transactions are completely isolated) to more relaxed levels (allowing some degree of concurrency).

Example:

Two transactions are executed simultaneously:

1. Transaction 1 reads a product's stock quantity.

2. Transaction 2 updates the stock quantity.

Isolation ensures that Transaction 1 sees a consistent view of the stock quantity, regardless of Transaction 2's operations. If Transaction 1 reads the stock before Transaction 2 updates it, Transaction 1 will not see any intermediate or partial results.

### *4. Durability*

Definition: Durability ensures that once a transaction is committed, its changes are permanently recorded in the database, even in the case of a system crash or failure. The effects of a committed transaction are not lost, and the database must be able to recover to the state afterthe transaction was completed.

Example:

After committing a transaction that updates an employee's salary, the change should be preserved in the database. Even if the system crashes immediately after the commit, the updated salary must be retained when the system restarts.

What is a Database?

A database is an organized collection of data that can be easily accessed, managed, and updated. It is designed to store large amounts of information efficiently and is typically managed using a Database Management System (DBMS). Databases are used in various applications such as e-commerce, banking, healthcare, and social media to ensure data integrity, security, and **availability.**

MODULE 1

Three-Schema Architecture

The Three-Schema Architecture, proposed by the ANSI/SPARC committee, is a framework for designing and managing databases. It separates a database into three levels to provide abstraction, flexibility, and independence between the physical storage of data and the user’s view of the data.

1. External Schema (User View):

- Purpose: Represents how individual users or applications interact with the database.
- Description:
  - Each user sees only the data relevant to their needs.
  - Provides different views of the database tailored to specific roles or use cases.
- Example:
  - A payroll department sees employee salaries and tax data, while HR sees personal details like addresses and job titles.
- Key Feature: Customizable views for different users without affecting other parts of the database.

2. Conceptual Schema (Logical Level):

- Purpose: Represents the logical structure of the entire database, independent of physical storage details.
- Description:
  - Defines entities, relationships, constraints, and data integrity rules.
  - Ensures consistency by providing a unified logical model for all external views.
- Example:
  - The database contains entities like Employee, Department, and their relationships, such as "Employee works in Department."
- Key Feature: Centralized representation of data, ensuring **data consistency across all user views.**

3. Internal Schema (Physical Level):

- Purpose: Describes how data is physically stored in the database.
- Description:
  - Focuses on storage structures, file organization, indexing, and access methods.
  - Handles performance optimization for data retrieval and storage.
- Example:
  - Data is stored in indexed tables, rows, or files on a disk.
- Key Feature: Storage management and efficiency.

MODULE 1

WHAT ARE THE ADVANTAGES OF USING DBMS APPORACH EXPLAIN

Data Redundancy Control

- Explanation: DBMS reduces data redundancy by ensuring a centralized repository of data, where information is stored only once.
- Example: In a university database, instead of storing a student's details in multiple files for admissions, grades, and library records, the data is centralized, avoiding duplication.

2. Data Consistency

- Explanation: With reduced redundancy, the DBMS ensures that data remains consistent across all uses.
- Example: If a student’s address is updated in one part of the system, it automatically reflects in all relevant areas (e.g., library and exam systems).

3. Data Integrity

- Explanation: DBMS enforces data integrity rules, ensuring that data is accurate and reliable.
- Example: A DBMS can ensure that an employee's salary is always a positive number or that an order date cannot be set in the future.

4. Data Security

- Explanation: DBMS provides robust mechanisms to restrict unauthorized access to sensitive data.
- Example: Employees in a payroll department may access salary **information, while other employees**

5. Improved Data Access and Sharing

- Explanation: DBMS allows multiple users to access and share data concurrently.
- Example: In a bank, tellers and ATM systems can simultaneously access a customer’s account balance.

6. Data Abstraction

- Explanation: DBMS hides the complexity of how data is stored, providing simplified interfaces for users to access data.
- Example: A user can query the database using SQL without needing to know the underlying file structure.

7. Data Independence

- Explanation: DBMS provides logical and physical independence, meaning changes to the data's structure or storage do not affect applications.
- Example: If the storage format of customer data changes, applications querying the data remain unaffected.

8. Enhanced Decision-Making

- Explanation: DBMS enables complex queries and reporting for better analysis.
- Example: Managers can generate sales trends or customer behavior reports for strategic decision-making.

9. Backup and Recovery

- Explanation: DBMS includes automated backup and recovery mechanisms to prevent data loss in case of failures.
- Example: In case of a server crash, DBMS can recover the last consistent state of the database.

MODULE 3

What is Normalization?

Normalization is a process in database design used to organize data to reduce redundancy and improve data integrity. It involves structuring a database into tables and defining relationships between them to ensure that data dependencies are logical and stored efficiently.

Normalization is typically performed in stages called normal forms (NFs), each addressing specific issues related to redundancy and dependency. The most common are 1NF (First Normal Form), 2NF (Second Normal Form), and 3NF (Third Normal Form).

1NF: First Normal Form

- **Definition: A table is in 1NF if:**
  1. **All columns contain atomic (indivisible) values.**
  2. **Each column contains values of a single type.**
  3. **Each record (row) is unique and identifiable by a primary key.**
- **Example (Unnormalized Table):**

Student_ID	Student_Name	Courses
101	Alice	Math, Science
102	Bob	History, English

  - **The Courses column contains multiple values, violating 1NF.**
- **1NF Table:**

Student_ID	Student_Name	Course
101	Alice	Math
101	Alice	Science
102	Bob	History
102	Bob	English

2NF: Second Normal Form

- Definition: A table is in 2NF if:
  1. It is in 1NF.
  2. All non-key attributes are fully functionally dependent on the entire primary key.
     - Partial dependency (where a non-key attribute depends on part of a composite key) must be removed.
- Example (1NF Table):

Student_ID	Course	Instructor
101	Math	Dr. Smith
101	Science	Dr. Adams
102	History	Dr. Green
102	English	Dr. Adams

  - Composite Primary Key: (Student_ID, Course)
  - Issue: Instructor depends only on Course, not on the entire composite key (Student_ID, Course).
- 2NF Tables: Table 1: Student-Course Relationship

Student_ID	Course
101	Math
101	Science
102	History
102	English

- Table 2: Course-Instructor Relationship

Course	Instructor
Math	Dr. Smith
Science	Dr. Adams
History	Dr. Green
English	Dr. Adams

3NF: Third Normal Form

- Definition: A table is in 3NF if:
  1. It is in 2NF.
  2. There are no transitive dependencies (non-key

attributes depending on other non-key attributes).

- Example (2NF Table):

Student_ID	Course	Instructor	Dept
101	Math	Dr. Smith	Math Dept
101	Science	Dr. Adams	Science Dept
102	History	Dr. Green	History Dept
102	English	Dr. Adams	English Dept

  - Issue: Dept depends on Instructor, not directly on Student_ID or Course. This creates a transitive dependency.
- 3NF Tables: Table 1: Student-Course Relationship

Student_ID	Course
101	Math
101	Science
102	History
102	English

- Table 2: Course-Instructor Relationship

Course	Instructor
Math	Dr. Smith
Science	Dr. Adams
History	Dr. Green
English	Dr. Adams

- Table 3: Instructor-Dept Relationship

Instructor	Dept
Dr. Smith	Math Dept
Dr. Adams	Science Dept
Dr. Green	History Dept

  - Transitive dependency is removed by splitting data into separate tables.