DATABASE FUNDAMENTALS

Database concept and architecture

Databases are structured systems used to store, manage, and retrieve data efficiently. The architecture of a database refers to its design and structure, which determines how data is organized, stored, and accessed. There are several key concepts and components involved in database systems:

1. Database Management System (DBMS):

A DBMS is software that allows users to define, create, maintain, and manage databases. It provides an interface between users, applications, and the database. Examples include MySQL, PostgreSQL, Oracle, and Microsoft SQL Server.

2. Database Models:

This refers to the way data is structured within the database. Common types include:

- **Relational Database Model**: Data is stored in tables (relations) with rows (records) and columns (attributes). SQL (Structured Query Language) is used to interact with relational databases.
- **NoSQL Databases**: These are used for more flexible, scalable data storage and include key-value stores, document-based databases (like MongoDB), column-family stores (like Cassandra), and graph databases (like Neo4j).
- **Object-Oriented Database**: Data is stored as objects, similar to the way data is handled in object-oriented programming.

3. DBMS Architecture:

DBMS architecture typically involves three levels:

- **Internal Level (Physical Level)**: Describes how data is stored physically on the storage medium (e.g., hard drives, SSDs). This includes file structures, indexing, and access methods.
- **Conceptual Level**: This is the logical view of the entire database. It defines the structure of the data, relationships, constraints, and operations allowed, without worrying about the physical storage details.
- **External Level (View Level)**: This defines how individual users or applications see the data. A single database may have multiple views tailored to the needs of different users.

4. Schema:

A schema is the blueprint or structure of the database. It defines how data is organized and how relationships among data are managed. It includes definitions of tables, fields, data types, and relationships.

- **Physical Schema**: Defines the physical storage of data.
- **Logical Schema**: Defines the logical view of the database (independent of physical considerations).

5. Data Independence:

- **Physical Data Independence**: The ability to change the physical storage of data without affecting the logical structure.
- **Logical Data Independence**: The ability to change the logical schema without affecting the application programs.

6. Data Integrity:

Refers to the accuracy, consistency, and reliability of data within the database. It is maintained through:

- **Entity Integrity**: Ensures that each record in a table is unique (usually with a primary key).
- **Referential Integrity**: Ensures that relationships between tables are consistent (e.g., foreign keys).
- **Domain Integrity**: Ensures that data values in a column fall within a valid set of values (e.g., constraints on data types).

7. Normalization:

This is the process of organizing data in a way that reduces redundancy and dependency. The goal is to minimize duplication of data and ensure that data is logically stored. It involves dividing large tables into smaller ones and using relationships (foreign keys) to link them.

8. Transactions:

A transaction is a sequence of operations performed as a single unit of work. It is essential for maintaining data consistency, especially in concurrent systems. The **ACID properties** are key to ensuring the reliability of transactions:

- **Atomicity**: A transaction is either fully completed or fully rolled back.
- **Consistency**: A transaction must leave the database in a consistent state.
- **Isolation**: Transactions should not interfere with each other.
- **Durability**: Once a transaction is committed, it is permanent.

9. Indexes:

Indexes are data structures that improve the speed of data retrieval operations. They work similarly to the index of a book and are often used to speed up searches on a database, especially for large datasets.

10. Query Processing and Optimization:

Databases typically use SQL for querying data. The DBMS optimizes queries to ensure efficient execution, using techniques like indexing, query rewriting, and execution plan generation.

11. Concurrency Control:

Concurrency control ensures that multiple transactions can be executed simultaneously without interfering with each other, preserving the consistency of the database. Methods like locking, timestamps, and multiversion concurrency control (MVCC) are commonly used.

12. Backup and Recovery:

These mechanisms ensure that data is not lost in case of system failures. Regular backups and recovery procedures are vital for maintaining database availability and integrity.

Types of Database Architectures:

- **Single-tier Architecture**: The DBMS and user interface run on the same machine. It is a simple system used for small-scale applications.
- **Two-tier Architecture**: The DBMS and the user interface are separated. The client sends requests to the server where the DBMS processes them.
- **Three-tier Architecture**: The DBMS is placed on a server, and the user interface and business logic are separated. This architecture is common in web applications, where the client (browser) interacts with the server (via web services), which interacts with the database.

13. Cloud Databases:

Cloud databases are hosted on cloud platforms and offer scalable, on-demand storage. These databases can be relational (e.g., Amazon RDS, Google Cloud SQL) or NoSQL (e.g., Amazon DynamoDB, MongoDB Atlas).

Understanding these concepts and the architecture behind them is crucial for designing efficient, scalable, and reliable database systems.

RDBMS

RDBMS stands for **Relational Database Management System**. It is a type of DBMS (Database Management System) that stores data in a structured format using rows and columns, which are organized into tables. An RDBMS uses the **relational model** to manage and query data, making it one of the most widely used types of databases.

Here’s a breakdown of the core concepts and features of an RDBMS:

Key Features of RDBMS:

1. **Tables**:
    - Data is stored in tables (also called relations).
    - Each table consists of rows (records) and columns (attributes).
    - Tables have a fixed schema, meaning that each column has a predefined data type (e.g., integer, varchar).
2. **Primary Key**:
    - A primary key is a unique identifier for each record in a table. It ensures that each record can be uniquely identified.
    - A primary key can consist of one or more columns.
    - Example: In a `Customers` table, `CustomerID` might be the primary key.
3. **Foreign Key**:
    - A foreign key is a column (or set of columns) in one table that refers to the primary key in another table.
    - It creates a relationship between two tables.
    - Example: In an `Orders` table, `CustomerID` could be a foreign key that links to the `CustomerID` in the `Customers` table.
4. **SQL (Structured Query Language)**:
    - SQL is the standard language used to interact with relational databases. It is used to query, insert, update, and delete data in the tables.
    - Common SQL operations include:
        - **SELECT**: Retrieve data
        - **INSERT**: Add new data
        - **UPDATE**: Modify existing data
        - **DELETE**: Remove data
5. **Normalization**:
    - RDBMSs rely on **normalization**, which is the process of organizing data to reduce redundancy and improve data integrity.
    - The process involves dividing large tables into smaller, related tables and using **foreign keys** to link them.
    - There are several normal forms (1NF, 2NF, 3NF, etc.), with each level aiming to reduce anomalies and improve database design.
6. **Data Integrity**:
    - **Entity Integrity**: Ensures that each record in a table has a unique identifier (primary key).
    - **Referential Integrity**: Ensures that foreign keys correctly refer to existing records in related tables, maintaining consistency across the database.
    - **Domain Integrity**: Ensures that the values in a column fall within a valid set or range (e.g., using constraints like `NOT NULL`, `CHECK`, etc.).
7. **ACID Properties**:
    - RDBMSs follow the **ACID** properties to ensure reliable transaction processing:
        - **Atomicity**: Transactions are all-or-nothing (either all operations succeed, or none of them do).
        - **Consistency**: The database moves from one valid state to another, ensuring data integrity.
        - **Isolation**: Transactions are isolated from one another, meaning the result of one transaction is not visible to others until it's completed.
        - **Durability**: Once a transaction is committed, the changes are permanent, even if the system crashes.
8. **Indexes**:

    - RDBMSs use indexes to speed up data retrieval. Indexes are created on one or more columns in a table to allow quick searching and sorting.
    - Example: An index on the `CustomerID` column in the `Customers` table allows faster lookup when querying customers by their ID.
9. **Relationships**:
    - RDBMSs support different types of relationships between tables:
        - **One-to-One**: A record in one table is related to one and only one record in another table.
        - **One-to-Many**: A record in one table is related to multiple records in another table.
        - **Many-to-Many**: Multiple records in one table are related to multiple records in another table (often implemented through a junction table).

Common RDBMSs:

1. **MySQL** – An open-source RDBMS widely used for web applications.
2. **PostgreSQL** – An open-source, advanced RDBMS known for its robustness and extensibility.
3. **Oracle Database** – A commercial RDBMS known for scalability and enterprise features.
4. **Microsoft SQL Server** – A popular RDBMS developed by Microsoft, commonly used in corporate environments.
5. **SQLite** – A lightweight, serverless RDBMS often used for embedded systems and mobile applications.

Advantages of RDBMS:

1. **Data Integrity and Accuracy**: By enforcing rules like primary keys and foreign keys, RDBMSs ensure that data is consistent and reliable.
2. **Flexibility**: RDBMSs allow complex queries using SQL, enabling sophisticated data manipulation and retrieval.
3. **Scalability**: They can handle large datasets, especially when optimized properly (e.g., with indexing).
4. **Security**: Most RDBMSs offer robust security features, including user authentication and authorization, encryption, and access control.

Disadvantages of RDBMS:

1. **Performance Issues**: As the size of the database grows, RDBMSs can experience performance issues, especially if not properly indexed or if queries are not optimized.
2. **Complexity**: Designing and maintaining an RDBMS can become complex, especially with large-scale or highly relational data models.
3. **Not Ideal for Unstructured Data**: RDBMSs are designed for structured data; they may not be the best choice for applications involving unstructured data like text, images, or multimedia.

What is Table in Database?

In a **database**, a **table** is a collection of data organized in rows and columns, where each row represents a record (or tuple) and each column represents a specific attribute or field of the data. Tables are the fundamental building blocks in a relational database, and they help store structured data in an organized and easily accessible way.

Structure of a Table:

- **Rows (Records)**:
  - Each row in a table represents a single, distinct record or data entry. A row contains specific values for each column.
  - For example, in a **Customers** table, each row would represent a different customer.
- **Columns (Fields or Attributes)**:
  - Columns represent the types of data or attributes that you want to store for each record. Each column is assigned a name and a data type (such as integer, text, or date).
  - For example, in the **Customers** table, columns might include **CustomerID**, **Name**, **Email**, **Phone Number**, etc.

Key Points About Tables in a Database:

1. **Unique Table Names**: Each table in a database must have a unique name to identify it. For example, a database may contain a table called **Employees**, and another table called **Orders**.
2. **Primary Key**:
   - A primary key is a column (or a combination of columns) that uniquely identifies each record in a table.
   - For example, **CustomerID** can be the primary key in the **Customers** table, ensuring that each customer has a unique identifier.
   - Primary keys help maintain data integrity by preventing duplicate records.
3. **Foreign Key**:
   - A foreign key is a column that links to the primary key in another table.
   - It helps establish relationships between tables. For example, in an **Orders** table, the **CustomerID** might be a foreign key that refers to the **CustomerID** in the **Customers** table.
4. **Data Types**:
   - Each column has a specific data type that defines what kind of data can be stored in it (e.g., `INT`, `VARCHAR`, `DATE`, `BOOLEAN`).
   - This helps ensure that the data stored in the table is consistent and valid.
5. **Constraints**:
   - Constraints define rules and conditions for data in the table. For example:
     - **NOT NULL** ensures that a column cannot have a NULL value.
     - **UNIQUE** ensures that all values in a column are different.
     - **CHECK** allows you to specify a condition for data (e.g., age must be greater than 18).
6. **Indexes**:
   - Indexes are often created on tables to improve the performance of query operations (like searching for specific values in columns).

- They help speed up data retrieval, but they can also slightly slow down data insertion and updates.

SQL Basic

SQL (Structured Query Language) is the standard language used to interact with relational databases. It is used for querying, inserting, updating, and deleting data, as well as managing database structures. Here's a guide to the basic SQL commands and concepts.

Basic SQL Commands:

1. **SELECT**: Retrieves data from a database.
   - The `SELECT` statement is used to query data from one or more tables.
   - Syntax:

     ```
     SELECT column1, column2, ...
     FROM table_name;
     ```

   - Example:

     ```
     SELECT Name, Email
     FROM Customers;
     ```

     This will retrieve the `Name` and `Email` columns from the `Customers` table.

2. **INSERT INTO**: Adds new data to a table.
   - The `INSERT INTO` statement is used to insert new rows into a table.
   - Syntax:

     ```
     INSERT INTO table_name (column1, column2, ...)
     VALUES (value1, value2, ...);
     ```

   - Example:

     ```
     INSERT INTO Customers (CustomerID, Name, Email)
     VALUES (1, 'John Doe', 'john.doe@example.com');
     ```

3. **UPDATE**: Modifies existing data in a table.
   - The `UPDATE` statement is used to modify one or more columns of an existing record.
   - Syntax:

     ```
     UPDATE table_name
     SET column1 = value1, column2 = value2, ...
     WHERE condition;
     ```

   - Example:

     ```
     UPDATE Customers
     SET Email = 'john.doe@newdomain.com'
     ```

```
WHERE CustomerID = 1;
```

This will update the email address for the customer with `CustomerID` 1.

4. **DELETE**: Removes data from a table.
   - The `DELETE` statement is used to delete rows from a table.
   - Syntax:

```
DELETE FROM table_name
WHERE condition;
```

   - Example:

```
DELETE FROM Customers
WHERE CustomerID = 1;
```

This will delete the row where the `CustomerID` is 1.

5. **CREATE TABLE**: Creates a new table in the database.
   - The `CREATE TABLE` statement is used to define a new table and its columns.
   - Syntax:

```
CREATE TABLE table_name (
    column1 datatype,
    column2 datatype,
    ...
);
```

   - Example:

```
CREATE TABLE Customers (
    CustomerID INT PRIMARY KEY,
    Name VARCHAR(100),
    Email VARCHAR(100),
    PhoneNumber VARCHAR(15)
);
```

This creates a `Customers` table with four columns: `CustomerID`, `Name`, `Email`, and `PhoneNumber`.

6. **ALTER TABLE**: Modifies an existing table's structure.
   - The `ALTER TABLE` statement is used to add, delete, or modify columns in an existing table.
   - Syntax:

```
ALTER TABLE table_name
ADD column_name datatype;         -- Add a new column
ALTER TABLE table_name
DROP COLUMN column_name;          -- Drop an existing column
ALTER TABLE table_name
MODIFY COLUMN column_name datatype; -- Modify column definition
```

   - Example:

```
ALTER TABLE Customers
ADD DateOfBirth DATE;
```

7. **DROP TABLE**: Deletes an entire table from the database.
    - The `DROP TABLE` statement is used to remove a table, including its structure and data, permanently.
    - Syntax:

      ```
      DROP TABLE table_name;
      ```

    - Example:

      ```
      DROP TABLE Customers;
      ```

Basic SQL Clauses and Concepts:

1. **WHERE**: Filters records based on specified conditions.
    - The `WHERE` clause is used to filter records and specify conditions.
    - Example:

      ```
      SELECT * FROM Customers
      WHERE Name = 'John Doe';
      ```

2. **AND/OR**: Combines multiple conditions in a `WHERE` clause.
    - `AND`: Both conditions must be true.
    - `OR`: At least one condition must be true.
    - Example:

      ```
      SELECT * FROM Customers
      WHERE Name = 'John Doe' AND Email = 'john.doe@example.com';
      ```

3. **ORDER BY**: Sorts the result set by one or more columns.
    - The `ORDER BY` clause is used to sort the result set in ascending (`ASC`) or descending (`DESC`) order.
    - Example:

      ```
      SELECT * FROM Customers
      ORDER BY Name ASC;  -- Ascending order
      ```

4. **LIMIT**: Limits the number of records returned.
    - The `LIMIT` clause is used to specify the number of records to return.
    - Example:

      ```
      SELECT * FROM Customers
      LIMIT 5;
      ```

5. **DISTINCT**: Removes duplicate values in the result set.
    - The `DISTINCT` keyword is used to return only distinct (unique) values.
    - Example:

      ```
      SELECT DISTINCT Email FROM Customers;
      ```

6. **JOIN**: Combines rows from two or more tables based on a related column.
    - SQL supports several types of joins:
        - **INNER JOIN**: Returns only matching rows from both tables.
        - **LEFT JOIN**: Returns all rows from the left table, and matching rows from the right table.
        - **RIGHT JOIN**: Returns all rows from the right table, and matching rows from the left table.
        - **FULL JOIN**: Returns all rows when there is a match in one of the tables.

    Example using an **INNER JOIN**:

    ```
    SELECT Orders.OrderID, Customers.Name
    FROM Orders
    INNER JOIN Customers
    ON Orders.CustomerID = Customers.CustomerID;
    ```

    This will retrieve the `OrderID` and `Customer Name` for each order in the `Orders` table, where there is a matching `CustomerID` in the `Customers` table.

7. **GROUP BY**: Groups rows that have the same values in specified columns into summary rows.
    - The `GROUP BY` clause is often used with aggregate functions like `COUNT()`, `SUM()`, `AVG()`, `MAX()`, and `MIN()`.
    - Example:

        ```
        SELECT COUNT(*), Country
        FROM Customers
        GROUP BY Country;
        ```

        This will count the number of customers in each country.

8. **HAVING**: Filters groups after the `GROUP BY` operation.
    - The `HAVING` clause is used to filter groups based on aggregate values.
    - Example:

        ```
        SELECT COUNT(*), Country
        FROM Customers
        GROUP BY Country
        HAVING COUNT(*) > 10;
        ```

SQL Data Types:

- **INT**: Integer data (whole numbers).
- **VARCHAR(size)**: Variable-length string (text data).
- **CHAR(size)**: Fixed-length string.
- **DATE**: Date in `YYYY-MM-DD` format.
- **BOOLEAN**: Stores `TRUE` or `FALSE`.
- **FLOAT**: Floating-point numbers.
- **DECIMAL**: Exact numeric values.

Example of a Basic SQL Query:

Let's create a `Customers` table and perform a few basic operations.

1. **Create Table:**

```
CREATE TABLE Customers (
    CustomerID INT PRIMARY KEY,
    Name VARCHAR(100),
    Email VARCHAR(100),
    PhoneNumber VARCHAR(15)
);
```

2. **Insert Data:**

```
INSERT INTO Customers (CustomerID, Name, Email, PhoneNumber)
VALUES (1, 'John Doe', 'john.doe@example.com', '555-1234'),
       (2, 'Jane Smith', 'jane.smith@example.com', '555-5678');
```

3. **Query Data:**

```
SELECT * FROM Customers;
```

4. **Update Data:**

```
UPDATE Customers
SET PhoneNumber = '555-4321'
WHERE CustomerID = 1;
```

5. **Delete Data:**

```
DELETE FROM Customers
WHERE CustomerID = 2;
```

Normalization

Normalization in DBMS (Database Management System) is the process of organizing the data in a way that reduces redundancy (duplicate data) and ensures data integrity. The main goal of normalization is to separate data into logical units (tables) and establish relationships between them, thus minimizing the risk of data anomalies such as **insertion**, **update**, and **deletion** anomalies.

Normalization is accomplished by dividing large tables into smaller ones and defining relationships between them. This process also ensures that the database schema adheres to certain rules called **normal forms**.

Why Normalize a Database?

- **Reduce Redundancy**: Avoids storing the same data multiple times, which reduces the risk of data inconsistency.
- **Improve Data Integrity**: Ensures that data is logically stored, which helps maintain the accuracy and consistency of data.
- **Optimize Queries**: Well-normalized databases often perform better by reducing the need for duplicate data and making indexing more efficient.
- **Eliminate Anomalies**: Helps prevent anomalies like **update**, **insertion**, and **deletion** anomalies.

Types of Normal Forms (NF):

The process of normalization typically involves progressing through several **normal forms**. Each normal form has specific rules, and a table can be in one normal form, two normal forms, or up to **Boyce-Codd Normal Form (BCNF)**. Here's a brief overview of the most common normal forms:

---

1. First Normal Form (1NF)

A table is in **1NF** if it meets the following conditions:

- **Atomic Values**: All columns contain atomic (indivisible) values. This means no repeating groups or arrays in a column.
- **Unique Rows**: Every row in the table must be unique, i.e., no duplicate rows.
- **No Repeating Groups**: Each column must contain only one value per row.

Example of 1NF:

Consider the following table with student courses (not in 1NF):

StudentID	Name	Courses
1	John	Math, Science
2	Alice	History, Math

In this example, the `Courses` column contains multiple values, which violates the **atomicity rule**.

To convert this into **1NF**, we need to separate the multiple courses into individual rows:

StudentID	Name	Course
1	John	Math

StudentID	Name	Course
1	John	Science
2	Alice	History
2	Alice	Math

Now, each column contains atomic values, and each row is unique.

---

2. Second Normal Form (2NF)

A table is in **2NF** if it meets the following conditions:

- It is already in **1NF**.
- It **removes partial dependency**, meaning every non-key attribute is fully dependent on the **entire** primary key, not just part of it.

Example of 2NF:

Consider the following table (not in 2NF) with student course information:

StudentID	CourseID	Instructor	InstructorPhone
1	M101	Dr. Smith	123-4567
1	S102	Dr. Brown	234-5678
2	M101	Dr. Smith	123-4567

Here, the primary key is a composite key consisting of `StudentID` and `CourseID`. However, the `Instructor` and `InstructorPhone` depend only on `CourseID`, not on the entire primary key. This is a **partial dependency**.

To convert this into **2NF**, we need to create a new table for instructor information:

Students-Courses Table:

StudentID	CourseID
1	M101
1	S102

StudentID	CourseID
2	M101

Courses-Instructors Table:

CourseID	Instructor	InstructorPhone
M101	Dr. Smith	123-4567
S102	Dr. Brown	234-5678

Now, all non-key attributes are fully dependent on the entire primary key in each table.

---

3. Third Normal Form (3NF)

A table is in **3NF** if it meets the following conditions:

- It is already in **2NF**.
- It **removes transitive dependency**, meaning that non-key attributes must not depend on other non-key attributes.

Example of 3NF:

Consider the following table (not in 3NF):

StudentID	Name	CourseID	Instructor	InstructorPhone
1	John	M101	Dr. Smith	123-4567
2	Alice	S102	Dr. Brown	234-5678

Here, the `InstructorPhone` depends on `Instructor`, which is a non-key attribute. This is a **transitive dependency** because `InstructorPhone` depends on `Instructor`, and `Instructor` depends on `CourseID`.

To convert this into **3NF**, we need to remove the transitive dependency:

Students-Courses Table:

StudentID	CourseID
1	M101
2	S102

Courses-Instructors Table:

CourseID	Instructor
M101	Dr. Smith
S102	Dr. Brown

Instructors-Phone Table:

Instructor	InstructorPhone
Dr. Smith	123-4567
Dr. Brown	234-5678

Now, all non-key attributes are fully dependent on the primary key, and no transitive dependencies remain.

---

4. Boyce-Codd Normal Form (BCNF)

A table is in **BCNF** if it meets the following conditions:

- It is already in **3NF**.
- Every **determinant** (an attribute that determines another attribute) must be a **candidate key**.

This means that if a non-prime attribute (an attribute that is not part of any candidate key) determines another attribute, the table is not in BCNF.

Example of BCNF:

Consider the following table (not in BCNF):

StudentID	CourseID	Instructor
1	M101	Dr. Smith
2	M101	Dr. Smith
3	S102	Dr. Brown

Here, `Instructor` determines `CourseID`, but `Instructor` is not a candidate key. Therefore, the table violates BCNF.

To convert this into **BCNF**, we separate the `Instructor` into a separate table and relate it back to the `CourseID`:

Courses Table:

CourseID	Instructor
M101	Dr. Smith
S102	Dr. Brown

Students-Courses Table:

StudentID	CourseID
1	M101
2	M101
3	S102

Now, `Instructor` is dependent only on `CourseID`, and all determiners are candidate keys.

Higher Normal Forms (4NF, 5NF, etc.)

- **4NF (Fourth Normal Form)**: A table is in 4NF if it is in **BCNF** and has no **multi-valued dependencies** (i.e., no attribute can be dependent on a set of independent attributes).
- **5NF (Fifth Normal Form)**: A table is in 5NF if it is in **4NF** and has no **join dependencies** (a condition where data cannot be reconstructed by joining multiple tables).

What is Transaction in DBMS?

A **Transaction** in a **Database Management System (DBMS)** is a sequence of one or more SQL operations (like insert, update, delete, or select) that are executed as a single unit. A transaction is treated as a single, indivisible unit of work that either completely succeeds or completely fails.

Transactions are crucial for ensuring the **integrity** and **consistency** of the database, especially when multiple users or processes are accessing the database concurrently.

Key Characteristics of a Transaction (ACID Properties)

The behavior of a transaction is governed by the **ACID properties**, which stand for:

1. **Atomicity**:
    - **Atomicity** ensures that all the operations within a transaction are completed successfully. If any part of the transaction fails, the entire transaction is rolled back, and no changes are made to the database.
    - In simpler terms: A transaction is **all-or-nothing**. Either all operations are applied, or none of them are.

    Example: If a transaction involves transferring money from one account to another, either the debit and credit operations both happen successfully, or neither happens.

2. **Consistency**:
    - **Consistency** ensures that a transaction brings the database from one valid state to another valid state. The integrity constraints (like foreign keys, primary keys, and custom rules) are maintained before and after the transaction.
    - For example, a transaction that transfers money must not violate the balance constraints (e.g., no account can have a negative balance).
3. **Isolation**:
    - **Isolation** ensures that the operations of a transaction are not visible to other transactions until the transaction is complete. Even if multiple transactions are executing simultaneously, each transaction should appear as if it is the only transaction being processed.
    - This property helps avoid **concurrent transaction anomalies** like dirty reads, non-repeatable reads, and phantom reads.

    Example: If two users are transferring money at the same time, each user should see the account balance as if they are the only one performing the operation.

4. **Durability**:
    - **Durability** ensures that once a transaction is committed (completed), its changes are permanent, even if there is a system crash.

- The changes made by a committed transaction are saved to the database, and any failure or system crash after the commit will not affect the data.

Example: After a bank transaction has been completed, even if the database crashes immediately afterward, the changes (like the transfer of money) will still be stored and recovered after the system is restored.

Types of Transactions in DBMS:

1. **Read Transaction**:
   - A transaction that only retrieves data from the database. It does not modify the database in any way.
2. **Write Transaction**:
   - A transaction that modifies the database. This could involve operations like inserting, updating, or deleting data.

Transaction States:

A transaction can be in one of the following states during its lifecycle:

1. **Active**: The transaction is currently being executed.
2. **Partially Committed**: The transaction has executed all its operations, but it has not yet been committed to the database.
3. **Committed**: The transaction has been successfully completed, and all changes made by the transaction have been saved to the database.
4. **Aborted**: The transaction has failed, and all changes made by the transaction are rolled back (undone) to restore the database to its state before the transaction started.
5. **Failed**: If a transaction encounters an error and is unable to proceed, it is in a failed state until it is either rolled back or aborted.

Example of a Transaction:

Let's consider a simple transaction where money is transferred from one account to another.

Steps involved in the transaction:

1. **Start the transaction**.
2. **Check the balance** of the source account to ensure sufficient funds.
3. **Deduct the amount** from the source account.
4. **Add the amount** to the target account.

5. **Commit** the transaction if all steps are successful. If any error occurs, **rollback** the transaction.