DBMS Sem

Unit 1

Database Management Systems (DBMS) Notes

1. Database System Applications: A Historical Perspective

- **Early Data Management**: Initially, data was managed using flat file systems, which had limitations in terms of data integrity and complex querying.
- **Hierarchical Model**: Introduced as an early DBMS model where data is organized in a tree-like structure. Examples include IBM's Information Management System (IMS).
- **Network Model**: A more flexible model than the hierarchical model, allowing more complex relationships between data. Example: Integrated Data Store (IDS).
- **Relational Model**: Proposed by E.F. Codd in 1970, organizing data into tables and enabling powerful querying using SQL. This model is widely used today.
- **Modern Advances**: Development of NoSQL databases for unstructured data, and NewSQL databases for high-performance requirements.

2. File Systems versus a DBMS

Feature	File Systems	DBMS
Data Storage	Stores data in files and directories	Stores data in structured tables with relationships
Data Integrity	Limited support for data integrity constraints	Enforces data integrity with constraints (e.g., foreign keys)
Querying	Requires custom code for data retrieval and manipulation	Provides powerful querying language (e.g., SQL)
Concurrency Control	Basic file locking mechanisms	Advanced concurrency control with transactions
Data Management	Manual file management and organization	Automated data management with indexing and optimization
Backup and Recovery	Basic file backup options	Comprehensive backup and recovery solutions
Scalability	Limited scalability	Designed for scalability and can handle large datasets

3. The Data Model

- **Relational Model**: Data is organized into tables (relations) with rows (tuples) and columns (attributes). Relationships are defined using primary and foreign keys.
- **Entity-Relationship Model**: Uses entities (objects) and relationships (associations) to represent data and its connections. Often used in database design.
- **Object-Oriented Model**: Integrates object-oriented programming principles with database management, allowing data to be represented as objects.
- **Document Model**: Stores data in documents (e.g., JSON, BSON), often used in NoSQL databases like MongoDB.
- **Graph Model**: Represents data as nodes and edges, emphasizing relationships between data points. Used in graph databases like Neo4j.

4. Levels of Abstraction in a DBMS

- **Physical Level**: Describes the physical storage of data, including file organization and indexing. Focuses on how data is actually stored on disk.
- **Logical Level**: Defines what data is stored and the relationships among data entities. Provides a conceptual view of the database schema.
- **View Level**: Offers different views or subsets of the data tailored to specific user needs. Helps in simplifying data access for end-users.
- **External Schema**: Defines user-specific views and access controls, providing data from the perspective of different users or applications.
- **Conceptual Schema**: Represents the entire database structure, including entities, relationships, and constraints, without concern for physical storage details.

5. Data Independence

- **Logical Data Independence**: Ability to change the conceptual schema without affecting external views or application programs. Enhances flexibility in database design.
- **Physical Data Independence**: Ability to change the internal schema (storage details) without affecting the conceptual schema. Supports changes in storage technology or optimization without disrupting applications.
- **Impact on Applications**: Provides a stable interface for applications, allowing changes in database structure or storage without requiring changes to the applications.
- **Data Evolution**: Facilitates database evolution and scaling by separating data management from application development.
- **Implementation**: Achieved through abstraction layers in DBMS architecture, ensuring that data changes are managed transparently.

6. Structure of a DBMS

- **Database Engine**: Core component responsible for storing, retrieving, and updating data. Handles data access and manipulation.
- **Database Schema**: Defines the structure of the database, including tables, columns, data types, and relationships between tables.
- **Query Processor**: Interprets and executes queries, optimizes query performance, and ensures efficient data retrieval.
- **Transaction Manager**: Manages transactions to ensure data consistency and integrity. Handles operations like commit and rollback.
- **Storage Manager**: Manages physical storage of data, including file organization, indexing, and buffer management.
- **Metadata**: Stores information about the database schema, including data definitions and constraints.
- **Security Manager**: Handles user authentication, authorization, and access control to ensure secure data management.

Introduction to Database Design

Database Design and ER Diagrams

- **Database Design**: The process of defining the structure of a database, including tables, relationships, and constraints, to ensure efficient data storage and retrieval. Effective design minimizes redundancy and improves data integrity.
- **ER Diagrams (Entity-Relationship Diagrams)**: A visual representation of the database structure. ER diagrams depict entities, their attributes, and the relationships between entities. They help in conceptualizing and documenting the database schema before implementation.
- **Components of ER Diagrams**:
  - **Entities**: Objects or things in the real world that are distinguishable. For example, `Customer`, `Order`, `Product`.
  - **Relationships**: Associations between entities. For example, `Customer` places `Order`.
  - **Attributes**: Properties or characteristics of entities. For example, `Customer` might have attributes like `CustomerID`, `Name`, `Address`.

Entities, Attributes, and Entity Sets

- **Entities**: Represent real-world objects or concepts. Entities can be physical (e.g., `Employee`) or abstract (e.g., `Course`).
- **Attributes**: Characteristics or properties of entities. They describe the data that can be stored about an entity. For example:
  - **Simple Attribute**: An attribute that cannot be divided further (e.g., `Name`, `DateOfBirth`).

    - **Composite Attribute**: An attribute that can be divided into smaller sub-parts (e.g., `Address` with sub-parts `Street`, `City`, `ZipCode`).
    - **Derived Attribute**: An attribute whose value can be derived from other attributes (e.g., `Age` derived from `DateOfBirth`).
    - **Multi-valued Attribute**: An attribute that can have multiple values (e.g., `PhoneNumbers` for a `Customer`).
- **Entity Sets**: A collection of similar entities. All entities in an entity set have the same attributes but may have different values. For example, an `Employee` entity set includes all employees, each with attributes like `EmployeeID`, `Name`, etc.

Relationships and Relationship Sets

- **Relationships**: Represent associations or interactions between entities. For example, `Enrolls` might represent a relationship between `Student` and `Course`.
- **Relationship Sets**: A collection of similar relationships. Each relationship set includes multiple relationships between the same sets of entities. For example, an `Enrolls` relationship set includes all enrollments of students in various courses.
- **Types of Relationships**:
    - **One-to-One (1:1)**: Each entity in one set is associated with at most one entity in the other set (e.g., `Person` and `Passport`).
    - **One-to-Many (1:N)**: Each entity in one set is associated with multiple entities in the other set (e.g., `Department` and `Employees`).
    - **Many-to-Many (M:N)**: Entities in one set can be associated with multiple entities in another set and vice versa (e.g., `Students` and `Courses`).

Additional Features of the ER Model

- **Weak Entities**: Entities that do not have a unique identifier on their own and are identified by their relationship with another entity. For example, `OrderItem` might be a weak entity with `Order` as its identifying entity.
- **Generalization and Specialization**:
    - **Generalization**: The process of extracting common characteristics from multiple entities to create a generalized entity (e.g., `Person` might generalize `Employee` and `Customer`).
    - **Specialization**: The process of defining specialized entities from a general entity (e.g., `Employee` might be specialized into `FullTimeEmployee` and `PartTimeEmployee`).
- **Aggregation**: A higher-level abstraction that allows the representation of a relationship between a relationship set and an entity set. For example, an `Enrollment` relationship might involve both `Student` and `Course`, and an `Aggregate` might represent this combination.
- **Composite Entities**: Entities that are used to represent many-to-many relationships. For example, `Enrollment` might be a composite entity representing

the many-to-many relationship between `Student` and `Course`.

Conceptual Design With the ER Model

- **Conceptual Design**: The process of creating a high-level description of the database structure using ER diagrams. It focuses on identifying entities, relationships, and constraints without considering implementation details.
- **Steps in Conceptual Design**:
  - **Identify Entities and Relationships**: Determine what entities exist and how they are related.
  - **Define Attributes**: Specify the attributes for each entity and relationship.
  - **Create ER Diagram**: Draw the ER diagram to visualize the entities, relationships, and attributes.
  - **Refine and Validate**: Review the ER diagram to ensure it accurately represents the requirements and make adjustments as necessary.
  - **Convert to Logical Schema**: Translate the ER diagram into a logical schema suitable for implementation in a specific DBMS.

Unit 2

Introduction to the Relational Model

Integrity Constraints over Relations

- **Integrity Constraints**: Rules applied to ensure the accuracy and consistency of data in a relational database. They help maintain the reliability of data by enforcing certain conditions.
- **Types of Integrity Constraints**:
  - **Domain Constraints**: Ensure that attribute values fall within a specified domain or range (e.g., age must be between 0 and 120).
  - **Entity Integrity**: Guarantees that each tuple (row) in a table has a unique identifier (primary key) and that primary key values are not null.
  - **Referential Integrity**: Ensures that a foreign key value in one table must match a primary key value in another table or be null (e.g., a `CustomerID` in an `Order` table must exist in the `Customer` table).
  - **Key Constraints**: Ensure that each tuple in a table is uniquely identifiable by its primary key, and no two tuples can have the same primary key value.

Enforcing Integrity Constraints

- **Primary Key Constraints**: Enforced automatically by the DBMS to ensure that primary key values are unique and not null.
- **Foreign Key Constraints**: Enforced by the DBMS to maintain referential integrity by checking that foreign key values match primary key values or are null.

- **Check Constraints**: Enforced by the DBMS to ensure that attribute values meet specified conditions (e.g., `CHECK (salary > 0)`).
- **Unique Constraints**: Enforced to ensure that values in specified columns are unique across the table (e.g., a unique email address for each user).

Querying Relational Data

- **SQL (Structured Query Language)**: The standard language used to query and manipulate relational data.
  - **SELECT**: Retrieves data from one or more tables (e.g., `SELECT * FROM Employees WHERE Department = 'HR';`).
  - **JOIN**: Combines rows from two or more tables based on a related column (e.g., `INNER JOIN`, `LEFT JOIN`).
  - **AGGREGATE FUNCTIONS**: Perform calculations on data, such as `COUNT`, `SUM`, `AVG`, `MIN`, and `MAX`.
  - **GROUP BY**: Groups rows sharing a property so aggregate functions can be applied (e.g., `GROUP BY Department`).
  - **ORDER BY**: Sorts query results based on one or more columns (e.g., `ORDER BY Salary DESC`).

Logical Database Design

- **Logical Database Design**: The process of translating conceptual designs (often represented as ER diagrams) into a logical schema that can be implemented in a relational database.
- **Normalization**: The process of organizing data to reduce redundancy and improve data integrity. Involves dividing tables into smaller tables and defining relationships between them. Normal forms (e.g., 1NF, 2NF, 3NF) are used to achieve this.
- **Relational Schema**: Defines the structure of the database in terms of tables, columns, and relationships. Includes primary and foreign keys, constraints, and data types.

Introduction to Views

- **Views**: Virtual tables created by querying one or more tables. They provide a way to present data from the database in a specific format or to restrict access to certain data.
- **Creating Views**: Defined using the `CREATE VIEW` statement (e.g., `CREATE VIEW EmployeeView AS SELECT Name, Salary FROM Employees WHERE Department = 'HR';`).
- **Updating Views**: Some views can be updated if they meet certain criteria (e.g., updatable views should map to a single table and not include aggregations).
- **Dropping Views**: Use `DROP VIEW` to remove a view from the database (e.g., `DROP VIEW EmployeeView;`).

Destroying/Altering Tables and Views

- **Destroying Tables**:
  - **DROP TABLE**: Removes a table and all its data from the database (e.g., `DROP TABLE Employees;`).
- **Altering Tables**:
  - **ALTER TABLE**: Modifies the structure of an existing table. Operations include adding or dropping columns, changing data types, and adding constraints (e.g., `ALTER TABLE Employees ADD COLUMN DateOfBirth DATE;`).
- **Destroying Views**:
  - **DROP VIEW**: Removes a view from the database (e.g., `DROP VIEW EmployeeView;`).
- **Altering Views**:
  - **CREATE OR REPLACE VIEW**: Updates the definition of an existing view (e.g., `CREATE OR REPLACE VIEW EmployeeView AS SELECT Name, Salary, Department FROM Employees;`).

Relational Algebra, Tuple Relational Calculus, and Domain Relational Calculus

Relational Algebra

- **Relational Algebra**: A procedural query language used to retrieve and manipulate data in a relational database. It operates on relations (tables) and produces a new relation as the result.
- **Basic Operations**:
  - **Selection (σ)**: Filters rows based on a condition. Syntax: `σ(condition)(Relation)`. Example: `σ(Salary > 50000)(Employees)` retrieves employees with salaries greater than 50,000.
  - **Projection (π)**: Selects specific columns from a relation. Syntax: `π(column1, column2, ...)(Relation)`. Example: `π(Name, Salary)(Employees)` retrieves only the `Name` and `Salary` columns.
  - **Union (∪)**: Combines the results of two relations, excluding duplicates. Syntax: `Relation1 ∪ Relation2`.
  - **Intersection (∩)**: Retrieves rows that are present in both relations. Syntax: `Relation1 ∩ Relation2`.
  - **Difference (−)**: Retrieves rows present in the first relation but not in the second. Syntax: `Relation1 - Relation2`.
  - **Cartesian Product (×)**: Combines each row of the first relation with each row of the second relation. Syntax: `Relation1 × Relation2`.
  - **Join (⋈)**: Combines rows from two relations based on a common attribute. Example: `Employees ⋈ Departments` where `Employees.DepartmentID =`

`Departments.DepartmentID`.
    - **Rename (ρ)**: Renames the relation or its attributes. Syntax: `ρ(newName, oldName1/attribute1, oldName2/attribute2, ...)(Relation)`.

Tuple Relational Calculus

- **Tuple Relational Calculus (TRC)**: A non-procedural query language that specifies queries by describing the desired properties of the result tuples.
- **Syntax**: `{ T | Condition(T) }`, where `T` is a tuple variable and `Condition(T)` is a predicate describing the desired properties of tuples.
- **Example**:
    - **Find names of employees with a salary greater than 50,000**: `{ T.Name | Employee(T) ∧ T.Salary > 50000 }`.
    - **Explanation**: This retrieves the names of tuples `T` from the `Employee` relation where the `Salary` attribute of `T` is greater than 50,000.

Domain Relational Calculus

- **Domain Relational Calculus (DRC)**: A non-procedural query language that specifies queries by describing the desired properties of attribute values.
- **Syntax**: `{ (d1, d2, ..., dn) | Condition(d1, d2, ..., dn) }`, where `d1, d2, ..., dn` are domain variables representing attribute values, and `Condition` is a predicate describing the desired properties of these values.
- **Example**:
    - **Find names of employees with a salary greater than 50,000**: `{ (N) | ∃S (Employee(N, S) ∧ S > 50000) }`.
    - **Explanation**: This retrieves the domain values `N` where there exists a salary `S` such that the employee with name `N` has a salary `S` greater than 50,000.

Comparison

Aspect	Relational Algebra	Tuple Relational Calculus	Domain Relational Calculus
Type	Procedural	Non-procedural	Non-procedural
Query Specification	Specifies operations on relations	Specifies properties of tuples	Specifies properties of domain values
Result	Produces a new relation	Produces a set of tuples	Produces a set of tuples
Approach	Focuses on how to retrieve data	Focuses on what data to retrieve	Focuses on what data to retrieve
Expression Example	`π(Name)(σ(Salary > 50000)(Employees))`	`{ T | Employee(T) ∧`	`{ (N) | ∃S (Employee(N, S) ∧ S > 50000) }`

Aspect	Relational Algebra	Tuple Relational Calculus	Domain Relational Calculus
		`T.Salary > 50000 }`	

Unit 3

SQL: Queries, Constraints, and Triggers

Basic SQL Query

- **Form of Basic SQL Query**: The general structure of a SQL query is:

```sql
SELECT column1, column2, ...
FROM table_name
WHERE condition
ORDER BY column1, column2, ...;
```

- **Example**:

```sql
SELECT Name, Salary
FROM Employees
WHERE Department = 'HR'
ORDER BY Salary DESC;
```

  This query selects the `Name` and `Salary` of employees from the `Employees` table where the department is 'HR', and orders the results by salary in descending order.

UNION, INTERSECT, and EXCEPT

- **UNION**: Combines the result sets of two or more SELECT queries and removes duplicate rows.
  - **Syntax**:

```sql
SELECT column1, column2, ...
FROM table1
UNION
SELECT column1, column2, ...
FROM table2;
```

- **Example**:

```sql
SELECT Name FROM Employees
UNION
SELECT Name FROM Contractors;
```

Retrieves a combined list of names from both `Employees` and `Contractors`, excluding duplicates.

- **INTERSECT**: Returns the common rows from two SELECT queries.
  - **Syntax**:

```sql
SELECT column1, column2, ...
FROM table1
INTERSECT
SELECT column1, column2, ...
FROM table2;
```

  - **Example**:

```sql
SELECT Name FROM Employees
INTERSECT
SELECT Name FROM Consultants;
```

Retrieves names that are common to both `Employees` and `Consultants`.

- **EXCEPT**: Returns rows from the first SELECT query that are not present in the second SELECT query.
  - **Syntax**:

```sql
SELECT column1, column2, ...
FROM table1
EXCEPT
SELECT column1, column2, ...
FROM table2;
```

- **Example**:

```sql
SELECT Name FROM Employees
EXCEPT
SELECT Name FROM RetiredEmployees;
```

Retrieves names of employees who are not in the `RetiredEmployees` table.

Nested Queries

- **Nested Queries**: Queries within queries that are used to perform operations based on the result of another query.
  - **Syntax**:

```sql
SELECT column1
FROM table1
WHERE column2 IN (SELECT column2 FROM table2 WHERE
condition);
```

  - **Example**:

```sql
SELECT Name
FROM Employees
WHERE DepartmentID IN (SELECT DepartmentID FROM Departments
WHERE Location = 'New York');
```

Retrieves names of employees who work in departments located in 'New York'.

Aggregation Operators

- **Aggregation Operators**: Functions that perform calculations on a set of values and return a single value.
  - **COUNT()**: Returns the number of rows.
    - **Example**: `SELECT COUNT(*) FROM Employees;`
  - **SUM()**: Returns the sum of values.
    - **Example**: `SELECT SUM(Salary) FROM Employees;`
  - **AVG()**: Returns the average of values.
    - **Example**: `SELECT AVG(Salary) FROM Employees;`
  - **MIN()**: Returns the minimum value.
    - **Example**: `SELECT MIN(Salary) FROM Employees;`
  - **MAX()**: Returns the maximum value.
    - **Example**: `SELECT MAX(Salary) FROM Employees;`

NULL Values

- **NULL Values**: Represent missing or unknown data. They are distinct from empty strings or zero values.
  - **Handling NULLs**:
    - Use `IS NULL` or `IS NOT NULL` to check for NULL values.
    - **Example**: `SELECT Name FROM Employees WHERE ManagerID IS NULL;` retrieves employees without managers.
    - Use `COALESCE()` to provide default values for NULLs.
    - **Example**: `SELECT Name, COALESCE(Salary, 0) AS Salary FROM Employees;` replaces NULL salaries with 0.

Complex Integrity Constraints in SQL

- **Unique Constraints**: Ensure that all values in a column are unique across the table.
  - **Syntax**:

```sql
CREATE TABLE Employees (
  EmployeeID INT PRIMARY KEY,
  Email VARCHAR(255) UNIQUE
);
```

- **Check Constraints**: Enforce domain constraints on values.
  - **Syntax**:

```sql
CREATE TABLE Employees (
  EmployeeID INT PRIMARY KEY,
  Age INT CHECK (Age >= 18)
);
```

- **Foreign Key Constraints**: Ensure that a value in one table corresponds to a value in another table.
  - **Syntax**:

```sql
CREATE TABLE Orders (
  OrderID INT PRIMARY KEY,
  CustomerID INT,
  FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)
);
```

Triggers and Active Databases

- **Triggers**: Automatic actions performed in response to specific events on a table (e.g., INSERT, UPDATE, DELETE).
  - **Creating a Trigger**:
    - **Syntax**:

```sql
CREATE TRIGGER trigger_name
AFTER INSERT ON table_name
FOR EACH ROW
BEGIN
  -- Trigger logic
END;
```

    - **Example**:

```sql
CREATE TRIGGER after_employee_insert
AFTER INSERT ON Employees
FOR EACH ROW
BEGIN
  INSERT INTO AuditLog (Action, EmployeeID, Timestamp)
  VALUES ('INSERT', NEW.EmployeeID, NOW());
END;
```

Logs each insertion into the `Employees` table to an `AuditLog`.

- **Active Databases**: Databases that use triggers and rules to automatically respond to certain changes or events, providing more dynamic data management and integrity enforcement.

Schema Refinement

Problems Caused by Redundancy

- **Redundancy**: Occurs when the same piece of data is stored in multiple places, leading to potential issues such as:
  - **Update Anomalies**: Difficulty in ensuring that all instances of a piece of data are updated consistently. For example, changing a customer's address might require updates in multiple tables.
  - **Insertion Anomalies**: Difficulty in inserting new data without introducing inconsistencies. For instance, adding a new product might require information about the product's category, which may not yet exist.
  - **Deletion Anomalies**: Loss of data due to the removal of data from one table that is necessary for other records. For example, deleting an employee might also remove their department information if not managed properly.

Decompositions

- **Decomposition**: The process of breaking down a relation into smaller relations to reduce redundancy and improve data integrity. This is often done using normalization techniques.
- **Objective**: Ensure that decomposed relations are in a form that avoids redundancy and maintains dependencies.
- **Types of Decomposition**:
  - **Lossless Join Decomposition**: Ensures that no information is lost during decomposition. The original relation can be reconstructed by joining the decomposed relations.

Problems Related to Decomposition

- **Lossless Join**: Ensuring that decompositions do not lose information. The original relation should be able to be perfectly reconstructed from the decomposed relations.
- **Dependency Preservation**: Ensuring that all functional dependencies are preserved in the decomposed relations. Functional dependencies should be expressible in terms of the decomposed relations.

Reasoning About Functional Dependencies

- **Functional Dependencies (FDs)**: Relationships between attributes in a relation where one attribute uniquely determines another.
  - **Notation**: `X → Y`, meaning attribute set `X` functionally determines attribute set `Y`.
  - **Inference Rules**: Used to derive all functional dependencies from a given set.
    - **Armstrong's Axioms**:
      - **Reflexivity**: If `Y ⊆ X`, then `X → Y`.
      - **Augmentation**: If `X → Y`, then `XZ → YZ` for any `Z`.
      - **Transitivity**: If `X → Y` and `Y → Z`, then `X → Z`.

First Normal Form (1NF)

- **1NF**: A relation is in 1NF if all its attributes contain only atomic (indivisible) values.
  - **Requirements**: Each cell in a table should hold a single value, and each column should have a unique name.
  - **Example**: A table with multiple values in a single cell (e.g., a list of phone numbers) is not in 1NF.

Second Normal Form (2NF)

- **2NF**: A relation is in 2NF if it is in 1NF and all non-key attributes are fully functionally dependent on the primary key.
  - **Requirements**: There should be no partial dependency of any attribute on a part of a composite primary key.
  - **Example**: In a relation with a composite key `(StudentID, CourseID)`, attributes such as `StudentName` should depend on `StudentID` only, not on the entire composite key.

Third Normal Form (3NF)

- **3NF**: A relation is in 3NF if it is in 2NF and no transitive dependency exists, i.e., non-key attributes are not dependent on other non-key attributes.
  - **Requirements**: Every non-key attribute must depend only on the primary key.

    - **Example**: If `CourseInstructor` depends on `CourseID` and `CourseID` depends on `StudentID`, then `CourseInstructor` should be moved to a separate relation to achieve 3NF.

Boyce-Codd Normal Form (BCNF)

- **BCNF**: A relation is in BCNF if it is in 3NF and every determinant is a candidate key.
    - **Requirements**: For every functional dependency `X → Y`, `X` must be a superkey.
    - **Example**: A relation where `EmployeeID` determines `Department` but `Department` is not a candidate key might need to be decomposed to achieve BCNF.

Lossless Join Decomposition

- **Lossless Join Decomposition**: A decomposition of a relation into smaller relations that preserves the original relation's information. The decomposition should satisfy:
    - **Join Dependency**: The original relation should be reconstructible without loss of information by joining the decomposed relations.
    - **Example**: Decomposing `R(A, B, C)` into `R1(A, B)` and `R2(A, C)` should be done in such a way that the natural join of `R1` and `R2` results in `R`.

Multivalued Dependencies

- **Multivalued Dependencies (MVDs)**: Occur when one attribute determines a set of values for another attribute. They are used to handle scenarios where a relation contains multiple sets of values for a single key.
    - **Notation**: `X ↠ Y`, meaning `X` determines multiple values of `Y`.
    - **Example**: A relation with `StudentID` determining multiple `CourseIDs` and `ProjectIDs`.

Fourth Normal Form (4NF)

- **4NF**: A relation is in 4NF if it is in BCNF and has no multivalued dependencies.
    - **Requirements**: The relation should be decomposed to remove any multivalued dependencies.
    - **Example**: If a relation `StudentID ↠ Course` and `StudentID ↠ Project`, it should be decomposed into separate relations to handle each set of multivalued dependencies.

Fifth Normal Form (5NF)

- **5NF**: A relation is in 5NF if it is in 4NF and every join dependency is implied by the candidate keys.

- **Requirements**: The relation should be decomposed to remove any join dependencies that are not implied by candidate keys.
- **Example**: A relation where multiple candidate keys might result in multiple joins and decompositions to ensure that all joins are necessary.

Unit 4

Transaction Concepts

Transaction Concept

- **Transaction**: A sequence of operations performed as a single logical unit of work. Transactions ensure the database remains in a consistent state even in the presence of system failures or concurrent access.
- **ACID Properties**: Transactions are required to adhere to four key properties:
  - **Atomicity**: Transactions are indivisible units of work; they either complete entirely or have no effect.
  - **Consistency**: Transactions must transition the database from one consistent state to another.
  - **Isolation**: Transactions should not interfere with each other's execution.
  - **Durability**: Once a transaction is committed, its changes are permanent, even in the case of a system failure.

Transaction State

- **States of a Transaction**:
  - **Active**: The transaction is currently being executed.
  - **Partially Committed**: The transaction has executed its final operation but has not yet been committed.
  - **Failed**: The transaction has encountered an error and cannot proceed.
  - **Aborted**: The transaction has been rolled back, undoing all changes made.
  - **Committed**: The transaction has completed successfully and its changes are permanent.

Implementation of Atomicity and Durability

- **Atomicity**: Achieved through the use of logs and recovery mechanisms. The database system ensures that either all operations in a transaction are applied, or none are.
  - **Undo Logs**: Store information to reverse the effects of incomplete transactions.
  - **Redo Logs**: Store information to apply the effects of completed transactions.
- **Durability**: Achieved by ensuring that once a transaction is committed, its changes are stored in non-volatile memory.

    - **Write-Ahead Logging (WAL)**: Ensures that changes are written to a log before they are applied to the database.

Concurrent Executions

- **Concurrent Executions**: Multiple transactions can be executed simultaneously, which can lead to various issues such as interference and inconsistencies.
- **Challenges**:
    - **Lost Updates**: Transactions overwrite each other's updates.
    - **Temporary Inconsistency**: Transactions read inconsistent data due to concurrent updates.
    - **Uncommitted Data**: Transactions read data from other transactions that have not yet committed.

Serializability

- **Serializability**: Ensures that concurrent transactions yield the same result as if they were executed serially (one after the other).
- **Types**:
    - **Conflict-Serializability**: Transactions are serializable if they can be transformed into a serial schedule by swapping non-conflicting operations.
    - **View-Serializability**: Transactions are serializable if they produce the same final database state as a serial execution.

Recoverability

- **Recoverability**: Ensures that the database can be restored to a consistent state after a failure, considering the effects of transactions.
- **Types**:
    - **Recoverable Schedule**: A schedule where, if a transaction T1 is followed by T2, T1 must commit before T2.
    - **Strict Schedule**: A stricter form where transactions must be serializable and recoverable.

Implementation of Isolation

- **Isolation**: Achieved through concurrency control mechanisms to ensure that transactions do not interfere with each other.
- **Techniques**:
    - **Lock-Based Protocols**: Use locks to control access to data.
    - **Timestamp-Based Protocols**: Use timestamps to manage transaction execution order.
    - **Validation-Based Protocols**: Validate transactions at commit time to ensure they do not violate serializability.

Testing for Serializability

- **Testing**: Methods to determine if a schedule is serializable.
  - **Serialization Graph**: A directed graph where nodes represent transactions and edges represent dependencies. If the graph has no cycles, the schedule is serializable.
  - **Precedence Graph**: Used to check if a schedule is conflict-serializable by analyzing transaction dependencies.

Lock-Based Protocols

- **Lock-Based Protocols**: Control access to data using locks to ensure transactions do not interfere with each other.
  - **Types**:
    - **Binary Locks**: A simple form of locking where each data item can be either locked or unlocked.
    - **Exclusive Locks (X-Locks)**: Allow a transaction to read and write a data item.
    - **Shared Locks (S-Locks)**: Allow a transaction to read a data item but not modify it.
  - **Two-Phase Locking (2PL)**: Ensures serializability by dividing the execution into two phases – growing (acquiring locks) and shrinking (releasing locks).

Timestamp-Based Protocols

- **Timestamp-Based Protocols**: Assign timestamps to transactions to ensure serializability.
  - **Basic Timestamp Ordering Protocol**: Transactions are executed based on their timestamps, and conflicts are resolved based on these timestamps.
  - **Thomas' Write Rule**: Allows some transactions to be aborted if they are outdated (i.e., their updates are no longer relevant).

Validation-Based Protocols

- **Validation-Based Protocols**: Validate transactions at commit time to ensure they are serializable.
  - **Three-Phase Validation**:
    - **Read Phase**: Transactions read data without making changes.
    - **Validation Phase**: The system checks if the transaction can be committed without violating serializability.
    - **Write Phase**: The transaction updates the database if validation is successful.

Multiple Granularity

- **Multiple Granularity**: Allows locking at different levels of granularity (e.g., tuples, pages, tables) to balance between performance and concurrency.

    - **Intention Locks**: Indicate a transaction's intention to acquire a lock on a more granular level.

Recovery and Atomicity

- **Recovery and Atomicity**: Achieved through techniques such as logging, checkpoints, and recovery algorithms to ensure that transactions can be rolled back or redone in case of failure.
    - **Log-Based Recovery**: Uses logs to record changes made by transactions and provides mechanisms to undo or redo changes as needed.

Log-Based Recovery

- **Log-Based Recovery**: Records all changes made by transactions in a log, which is used to recover the database state after a failure.
    - **Write-Ahead Logging (WAL)**: Ensures that changes are logged before they are applied to the database.
    - **Checkpoints**: Periodically save the state of the database to reduce recovery time.

Recovery with Concurrent Transactions

- **Recovery with Concurrent Transactions**: Involves handling transactions that overlap in time while ensuring atomicity and consistency.
    - **Techniques**:
        - **Undo-Redo Logging**: Maintains logs for both undoing changes from incomplete transactions and redoing changes from committed transactions.
        - **Distributed Transactions**: Manages transactions that span multiple databases or systems, using protocols like Two-Phase Commit (2PC) to ensure consistency.

Unit 5

Data Storage and Indexing

Data on External Storage

- **External Storage**: Refers to storage systems that are separate from the main memory, such as hard drives or SSDs, used to store large amounts of data persistently.
- **Characteristics**:
    - **Persistence**: Data remains stored even after the system is powered off.
    - **Capacity**: Typically has much larger capacity compared to main memory.
    - **Access Time**: Generally slower than main memory access times.

File Organization and Indexing

- **File Organization**: The method used to arrange data in files to optimize access and performance.
  - **Sequential Organization**: Records are stored in a sequential order, suitable for applications where records are processed in a specific order.
  - **Heap (Unordered) Organization**: Records are stored in arbitrary order; suitable for applications with random access patterns.
- **Indexing**: A technique used to improve the speed of data retrieval operations by providing a quick way to locate data.
  - **Purpose**: Reduces the amount of data that needs to be scanned to find specific records.
  - **Types**:
    - **Cluster Indexes**
    - **Primary and Secondary Indexes**
    - **Hash-Based Indexing**
    - **Tree-Based Indexing**

Cluster Indexes

- **Cluster Index**: An index where the index order matches the physical order of the data rows.
  - **Characteristics**:
    - **Physical Organization**: Data is physically organized on the disk based on the cluster index.
    - **Efficiency**: Efficient for range queries as the data is stored contiguously.
  - **Example**: If you have a cluster index on the `Date` field of a table, the data will be stored in chronological order on disk.

Primary and Secondary Indexes

- **Primary Index**: An index on the primary key of a table.
  - **Characteristics**:
    - **Uniqueness**: Ensures that all values in the primary key column are unique.
    - **Single Index**: Typically, there is only one primary index per table.
  - **Example**: An index on the `EmployeeID` column if `EmployeeID` is the primary key.
- **Secondary Index**: An index on columns other than the primary key.
  - **Characteristics**:
    - **Non-unique**: May not enforce uniqueness.
    - **Multiple Indexes**: A table can have multiple secondary indexes.

- **Example**: An index on the `LastName` column for quick lookups by surname.

Index Data Structures

- **Hash-Based Indexing**: Uses a hash function to determine the location of a record.
  - **Characteristics**:
    - **Fast Access**: Provides constant time complexity for search operations.
    - **Limitations**: Not suitable for range queries or sorting.
  - **Example**: Hashing the `EmployeeID` to directly access the record.
- **Tree-Based Indexing**: Uses tree structures to index data.
  - **Characteristics**:
    - **Sorted Data**: Maintains data in a sorted order.
    - **Range Queries**: Efficient for range queries and ordered data access.
  - **Examples**:
    - **B+ Trees**
    - **B Trees**

Comparison of File Organizations

File Organization	Characteristics	Use Case
Sequential	Records stored in a fixed, sequential order.	Suitable for batch processing.
Heap	Records stored in arbitrary order.	Suitable for random access.
Indexed	Uses indexes to quickly locate records.	Suitable for high-speed queries.
Clustered	Data stored in the order of the index.	Suitable for range queries.

Intuitions for Tree Indexes

- **Tree Indexes**: Data is organized in a tree-like structure where each node contains keys and pointers to child nodes.
  - **Characteristics**:
    - **Balanced Trees**: Maintains balance to ensure logarithmic access times.
    - **Efficient Searching**: Allows efficient searching, insertion, and deletion.
  - **Examples**:
    - **B Trees**
    - **B+ Trees**

Indexed Sequential Access Methods (ISAM)

- **ISAM**: An indexing technique that combines indexing with sequential file access.
  - **Characteristics**:
    - **Static Index**: Index is built once and remains unchanged.
    - **Sequential Access**: Allows for efficient sequential access to records.
  - **Use Case**: Suitable for applications where data is mostly read and updated infrequently.

B+ Trees: A Dynamic Index Structure

- **B+ Trees**: A balanced tree data structure where all values are found at the leaf level, and internal nodes store only keys.
  - **Characteristics**:
    - **Balanced**: Ensures that all leaf nodes are at the same level.
    - **Efficient Range Queries**: Allows efficient range queries by traversing leaf nodes.
    - **Dynamic**: Supports dynamic insertion and deletion without rebalancing the entire tree.
  - **Structure**:
    - **Leaf Nodes**: Contain actual data or pointers to data.
    - **Internal Nodes**: Contain keys to guide the search and pointers to child nodes.