DBMS

UNIT 1

Database & Database Users

1. Database:
    - A database is an organized collection of data stored and accessed electronically.
    - It is designed to efficiently store, retrieve, and manage large amounts of structured information.
    - Databases are used to support various applications and enable efficient data manipulation and retrieval.
    - Common types of databases include relational databases, object-oriented databases, and NoSQL databases.

2. Database Users:
    - Database users are individuals or software applications that interact with the database system.
    - There are several types of database users, including:
      a. Database Administrators (DBAs): DBAs are responsible for managing and maintaining the database system. They perform tasks such as database design, installation, configuration, security management, and performance tuning.
      b. Database Developers: Database developers are responsible for designing and implementing the database schema and writing queries to manipulate and retrieve data from the database.
      c. End Users: End users are the individuals who use the database to perform specific tasks or access information. They may interact with the database through user-friendly interfaces or applications.
      d. Application Programmers: Application programmers develop software applications that interact with the database. They write code to connect to the database, execute queries, and process the retrieved data.
      e. System Analysts: System analysts are involved in analyzing the organization's requirements and designing the overall structure and functionality of the database system.
      f. Data Stewards: Data stewards are responsible for data governance, ensuring data quality, and enforcing data policies and standards.
      g. Executives: Executives use databases to obtain strategic information and make data-driven decisions for the organization.
    - Each type of user may have different levels of access and privileges based on their role and responsibilities within the organization.
    - Database systems provide mechanisms to control user access, maintain data integrity, enforce security, and manage user permissions.

Characteristics of the database systems

1. Data Independence:
    - Database systems provide data independence, which means that the database structure (schema) can be modified without affecting the applications or programs that use the data. There are two types of data

independence:
a. Logical Data Independence: The ability to modify the logical schema without changing the external schema or application programs.
b. Physical Data Independence: The ability to modify the physical schema without changing the logical schema or application programs.

2. Data Integration:
    - Database systems allow the integration of data from multiple sources and provide a unified view of the data. This eliminates data redundancy and inconsistencies that can occur in traditional file-based systems.
3. Data Sharing:
    - Database systems enable concurrent access to the data by multiple users and applications. This allows for data sharing and collaboration within an organization, improving productivity and data consistency.
4. Data Security:
    - Database systems provide mechanisms for ensuring data security and enforcing access control. Users can be granted specific privileges and permissions to restrict unauthorized access to sensitive data.
5. Data Integrity:
    - Database systems enforce data integrity constraints to maintain the accuracy and consistency of the data. These constraints can include primary key constraints, foreign key constraints, unique constraints, and other business rules defined by the database schema.
6. Data Atomicity, Consistency, Isolation, and Durability (ACID properties):
    - Database systems ensure ACID properties for reliable transaction processing:
      a. Atomicity: Transactions are treated as indivisible units of work, ensuring that either all the changes made by a transaction are committed or none of them are.
      b. Consistency: Transactions bring the database from one consistent state to another consistent state. The database remains in a valid state at all times.
      c. Isolation: Transactions execute in isolation from each other, preventing

interference and maintaining data integrity.
d. Durability: Once a transaction is committed, its changes are permanent and survive system failures.

7. Data Scalability and Performance:
    - Database systems offer scalability to handle increasing amounts of data and user load. They are designed to optimize query execution and provide mechanisms for indexing, query optimization, and caching to improve performance.
8. Data Backup and Recovery:
    - Database systems provide backup and recovery mechanisms to ensure data can be restored in case of hardware failures, software errors, or other unforeseen circumstances. Regular backups and transaction logs are maintained to enable recovery to a consistent state.

Concepts and Architecture of Database Systems:

1. Data Model:
    - A data model is a conceptual representation of the data structure, relationships, constraints, and semantics in a database. It provides a way to describe, organize, and manipulate data.
    - Commonly used data models include the relational model, hierarchical model, network model, and object-oriented model.
2. Schema:
    - A database schema defines the logical structure of a database. It specifies the tables, attributes, data types, relationships, and constraints for storing and organizing data.
    - There are two types of schemas:
      a. Logical Schema: Describes the database structure from a user's perspective, including tables, views, and relationships.
      b. Physical Schema: Describes how the data is physically stored in the database, including file organization, indexing, and storage structures.
3. Database Management System (DBMS):

- A DBMS is software that allows users to create, manipulate, and manage databases. It provides an interface between users and the underlying database, handling tasks such as data storage, retrieval, security, concurrency control, and query optimization.
- Common examples of DBMSs include Oracle, MySQL, Microsoft SQL Server, PostgreSQL, and MongoDB.

4. Three-Level Architecture:
    - The three-level architecture, also known as the ANSI/SPARC architecture, is a conceptual framework for organizing and designing database systems. It consists of the following levels:
      a. External Level (View Level): Represents the user's view of the database. It defines the user's perception of the data and provides a customized view of the database based on specific requirements.
      b. Conceptual Level (Logical Level): Represents the global or conceptual view of the entire database. It describes the overall structure, relationships, and constraints of the data in a database.
      c. Internal Level (Physical Level): Represents the physical storage and implementation details of the database. It deals with how the data is stored, accessed, and organized on the storage media.
5. Data Manipulation Language (DML):
    - DML is a language that allows users to manipulate and query the data in the database. It includes commands for inserting, updating, deleting, and retrieving data.
    - Examples of DML include SQL (Structured Query Language) for relational databases and NoSQL query languages like MongoDB's query language.
6. Data Definition Language (DDL):
    - DDL is a language used to define and modify the structure of the database. It includes commands for creating tables, defining relationships, specifying constraints, and altering the schema.
    - Examples of DDL include SQL statements such as CREATE, ALTER, and DROP.
7. Query Optimization:

- Query optimization is the process of selecting the most efficient execution plan for a given database query. The goal is to minimize the query's response time and resource usage by choosing the best access paths, join algorithms, and indexing strategies.

8. Transaction Management:
    - Transaction management ensures the integrity and consistency of the database. A transaction represents a unit of work that should be executed atomically and reliably.
    - ACID properties (Atomicity, Consistency, Isolation, Durability) ensure that transactions are processed reliably and maintain data integrity.
9. Instances:
    - an instance refers to a snapshot or specific state of a database at a given point in time. It represents the actual data stored in the database and includes the current values of all the data elements.

DBMS Architecture:

The architecture of a Database Management System (DBMS) refers to the overall structure and components that make up the system. It provides a framework for managing databases efficiently. Although specific implementations may vary, most DBMS architectures consist of the following key components:

1. User Interface:
    - The user interface component enables users to interact with the database system. It can include command-line interfaces, graphical user interfaces (GUIs), or application programming interfaces (APIs) that allow users to submit queries, retrieve data, and perform database operations.
2. Database Engine:
    - The database engine is the core component responsible for managing the actual database. It includes various subcomponents:
      a. Query Processor: The query processor analyzes and optimizes user queries, determining the most efficient execution plan.
      b. Storage Manager: The storage manager handles data storage and retrieval operations, including managing disk space, data buffering,

caching, and indexing.
c. Transaction Manager: The transaction manager ensures the ACID properties of transactions, coordinating concurrent access and recovery in case of failures.
d. Concurrency Control Manager: The concurrency control manager handles concurrent access to the database, ensuring data consistency and preventing conflicts between concurrent transactions.
e. Recovery Manager: The recovery manager handles database recovery in case of failures, ensuring that the database is restored to a consistent state.

3. Data Dictionary:
    - The data dictionary, also known as the metadata repository, stores metadata about the database. It includes information about tables, attributes, constraints, indexes, and relationships. The data dictionary is used by the DBMS to interpret and enforce the database schema.
4. Data Storage:
    - The data storage component is responsible for storing the actual data on disk or other storage media. It manages the physical representation of the database, organizing data into files or data structures optimized for efficient storage and retrieval.
5. Data Access Interfaces:
    - Data access interfaces provide ways for applications to interact with the database. This can include APIs, drivers, or libraries that enable programmers to connect to the DBMS, execute queries, and retrieve or modify data.

Data Independence:

Data independence refers to the ability to modify the database schema without impacting the application programs or user views that access the data. It provides a layer of abstraction that separates the logical structure of the database from the physical storage details. There are two types of data independence:

1. Logical Data Independence:
    - Logical data independence allows modifications to the logical schema (organization of data) without affecting the external schema (user views or

application programs) that access the data.

- For example, changes to the table structure, addition or removal of attributes, or modifications to relationships can be made at the logical level without requiring changes to the external views.

2. Physical Data Independence:
    - Physical data independence allows modifications to the physical storage details without affecting the logical or external schema. It enables changes to the storage structures, indexing mechanisms, or storage media without impacting the way data is accessed or manipulated.
    - For example, changing the file organization, reorganizing indexes, or migrating to a different storage system can be done at the physical level without requiring changes to the logical or external views.

Data independence provides flexibility and simplifies the maintenance and evolution of databases. It allows for modifications and optimizations at different levels without disrupting the applications or users interacting with the database.

Database Languages and Interfaces:

Database systems provide various languages and interfaces that allow users to interact with and manipulate the data stored in the database. These languages and interfaces serve different purposes and cater to different user roles and requirements. Here are some of the commonly used ones:

1. Structured Query Language (SQL):
    - SQL is a standardized language used for managing relational databases. It provides a comprehensive set of commands for defining database structure, querying data, modifying data, and managing database objects.
    - SQL is divided into several sublanguages, including Data Definition Language (DDL) for creating and modifying database schema, Data Manipulation Language (DML) for querying and modifying data, and Data Control Language (DCL) for managing user privileges and access control.
2. Procedural Language Extensions:
    - Many database systems provide procedural language extensions that allow users to write procedural code within the database. Examples include:

a. PL/SQL: Oracle's procedural language extension for SQL.
b. T-SQL: Microsoft SQL Server's procedural language extension for SQL.
c. PL/pgSQL: PostgreSQL's procedural language extension.

3. Application Programming Interfaces (APIs):
    - APIs provide a programming interface for developers to interact with databases programmatically. They offer language-specific libraries and methods to connect to the database, execute queries, retrieve data, and perform database operations.
    - Examples include:
      a. JDBC (Java Database Connectivity): API for connecting Java applications to databases.
      b.
      ADO.NET: API for connecting .NET applications to databases.
      c. ODBC (Open Database Connectivity): API for connecting applications to databases using a common interface.
4. Object-Relational Mapping (ORM) Frameworks:
    - ORM frameworks provide a higher-level abstraction for working with databases, mapping database tables to object-oriented classes. They handle the translation between object-oriented code and relational database operations, making database interactions more intuitive and efficient.
    - Examples include Hibernate (Java), Entity Framework (C#/.NET), and Django ORM (Python).
5. Command-Line Interfaces (CLI):
    - Command-line interfaces provide a text-based interface for interacting with the database. Users can enter commands directly into the terminal or console to perform database operations, execute queries, and view results.
6. Graphical User Interfaces (GUI):
    - GUI-based tools provide visual interfaces for managing and interacting with databases. They often offer features such as visual query builders, schema designers, data exploration, and administration capabilities.

- Examples include phpMyAdmin, SQL Server Management Studio, Oracle SQL Developer, and MySQL Workbench.

These languages and interfaces cater to different user roles and scenarios, ranging from database administrators and developers to end-users and analysts. Understanding and utilizing the appropriate language or interface based on your requirements is essential for effectively working with databases.

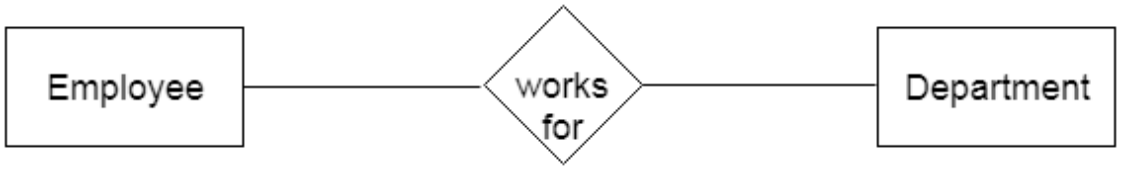
Data Modeling using the Entity-Relationship (ER) Approach:

The Entity-Relationship (ER) approach is a widely used method for data modeling in database systems. It provides a graphical representation of the database structure, focusing on the entities, their attributes, and the relationships between them. Here are the key components and concepts of the ER approach:

1. Entity:

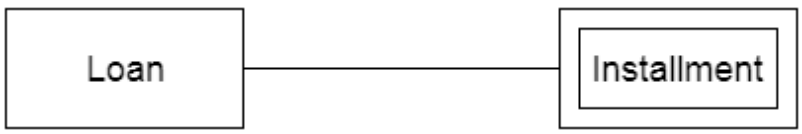
   An entity may be any object, class, person or place. In the ER diagram, an entity can be represented as rectangles.

   Consider an organization as an example- manager, product, employee, department etc. can be taken as an entity.



   a. Weak Entity

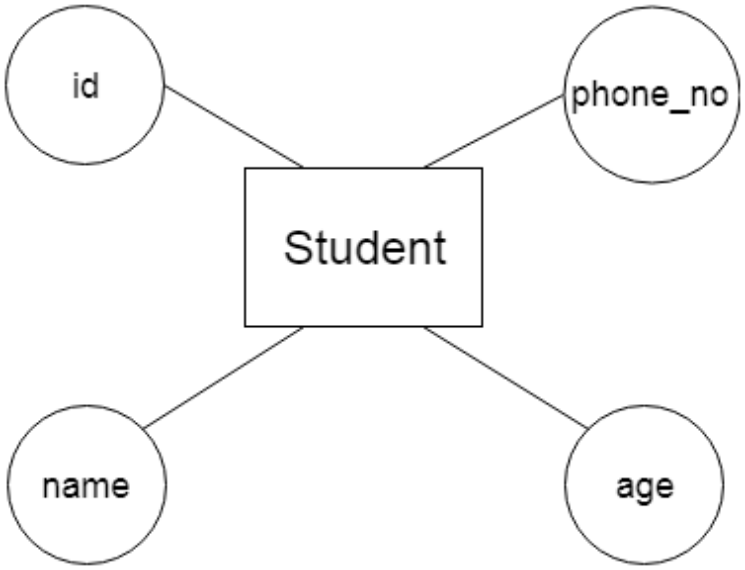
   An entity that depends on another entity called a weak entity. The weak entity doesn't contain any key attribute of its own. The weak entity is represented by a double rectangle.



2. Attribute:

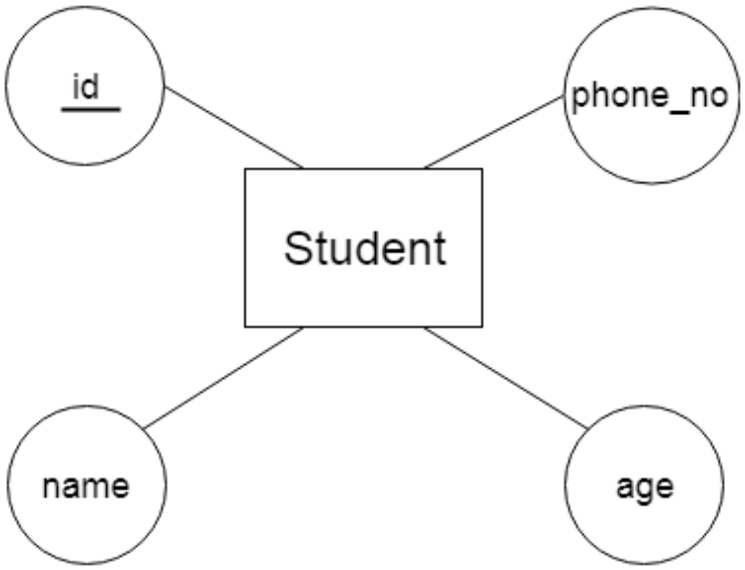
The attribute is used to describe the property of an entity. Eclipse is used to represent an attribute.

For example, id, age, contact number, name, etc. can be attributes of a student.



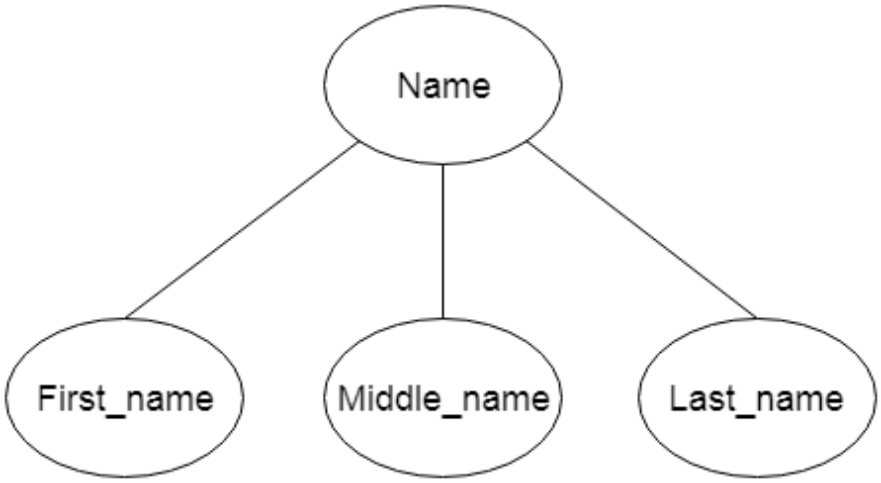
a. Key Attribute

The key attribute is used to represent the main characteristics of an entity. It represents a primary key. The key attribute is represented by an ellipse with the text underlined.



b. Composite Attribute

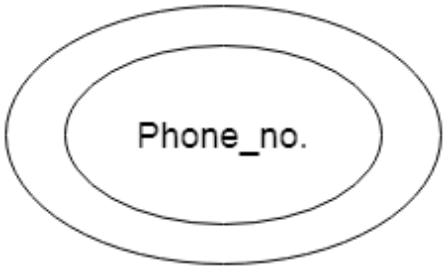
An attribute that composed of many other attributes is known as a composite attribute. The composite attribute is represented by an ellipse, and those ellipses are connected with an ellipse.



c. Multivalued Attribute

An attribute can have more than one value. These attributes are known as a multivalued attribute. The double oval is used to represent multivalued attribute.

For example, a student can have more than one phone number.



d. Derived Attribute

An attribute that can be derived from other attribute is known as a derived attribute. It can be represented by a dashed ellipse.

For example, A person's age changes over time and can be derived from another attribute like Date of birth.

3. Relationship:

A relationship is used to describe the relation between entities. Diamond or rhombus is used to represent the relationship.

Types of relationship are as follows:

a. One-to-One Relationship

When only one instance of an entity is associated with the relationship, then it is known as one to one relationship.

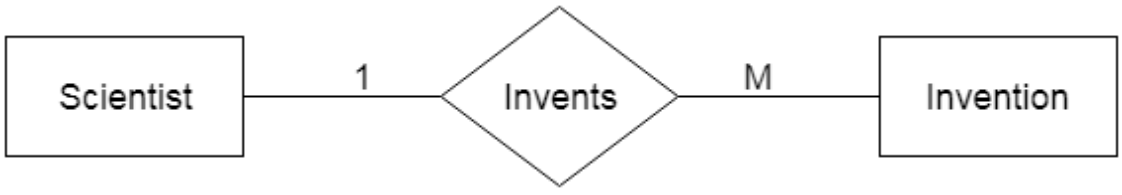
For example, A female can marry to one male, and a male can marry to one female.

AD

b. One-to-many relationship

When only one instance of the entity on the left, and more than one instance of an entity on the right associates with the relationship then this is known as a one-to-many relationship.

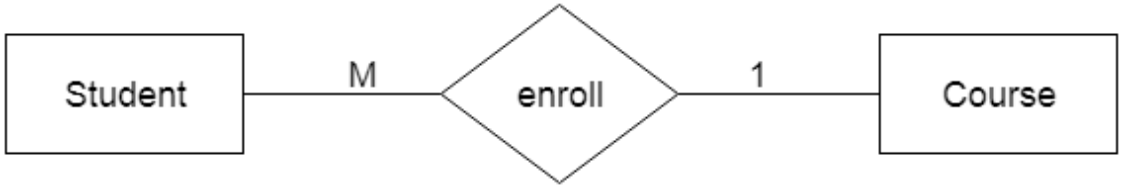
For example, Scientist can invent many inventions, but the invention is done by the only specific scientist.



c. Many-to-one relationship

When more than one instance of the entity on the left, and only one instance of an entity on the right associates with the relationship then it is known as a many-to-one relationship.

For example, Student enrolls for only one course, but a course can have many students.



d. Many-to-many relationship

When more than one instance of the entity on the left, and more than one instance of an entity on the right associates with the relationship then it is known as a many-to-many relationship.

For example, Employee can assign by many projects and project can have many employees.

4. Cardinality and Modality:
    - Cardinality defines the number of instances or occurrences of an entity that can be associated with another entity through a relationship. It specifies the minimum and maximum number of entities involved in a relationship. Common cardinality notations include 1, N, 0..1, and M.
    - Modality, also known as optionality, describes whether the participation of an entity in a relationship is mandatory (denoted by a solid line) or optional (denoted by a dashed line).
5. Primary Key:
    - A primary key is a unique identifier for each instance of an entity. It uniquely identifies each entity and is used to ensure data integrity and

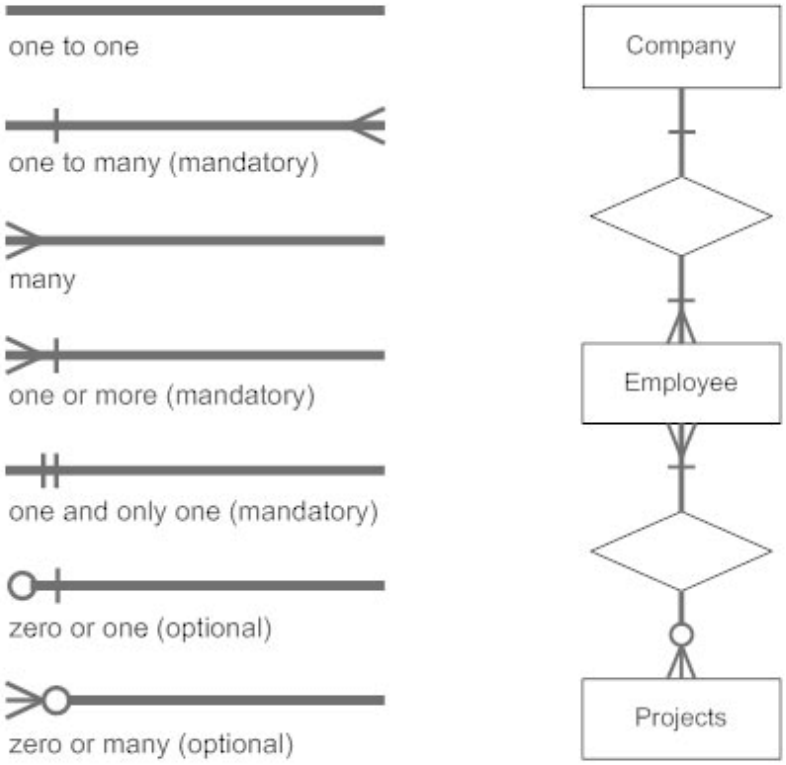
provide a means of referencing entities in relationships.

6. Foreign Key:
    - A foreign key is a reference to the primary key of another entity. It establishes a relationship between entities by linking attributes that represent the association between them.
7. ER Diagram:
    - An ER diagram visually represents the entities, attributes, relationships, and constraints in a database system. It provides a graphical representation of the data model, allowing for better understanding, communication, and design of the database structure.
8. Normalization:
    - Normalization is a process used to eliminate data redundancy and improve data integrity in a database. It involves breaking down a larger table into smaller, well-structured tables to minimize data duplication and dependency.

The ER approach provides a clear and intuitive way to model the structure of a database, capturing the entities, their attributes, and the relationships between them. It helps in designing an efficient and well-structured database schema.

Notation of ER diagram

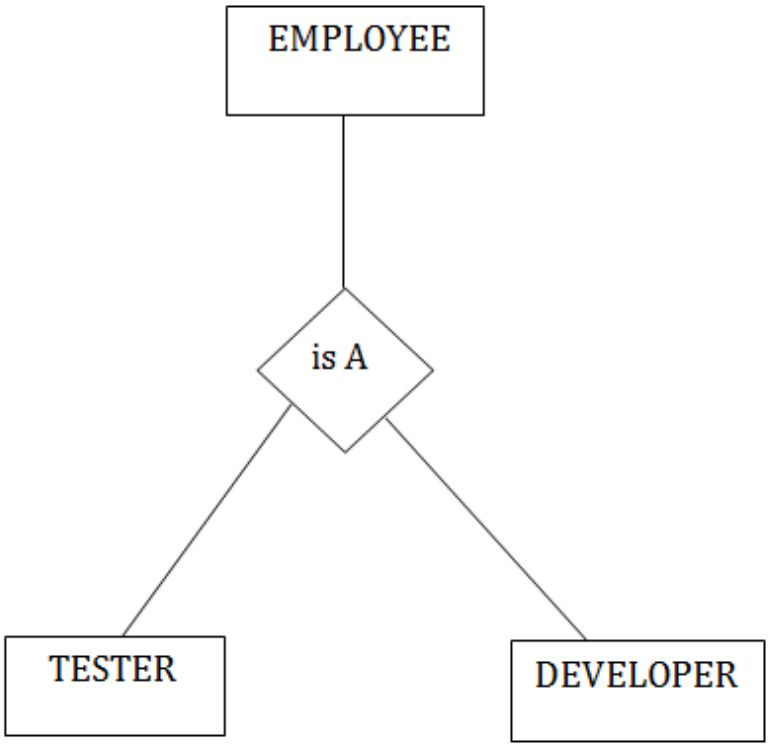
Database can be represented using the notations. In ER diagram, many notations are used to express the cardinality. These notations are as follows:



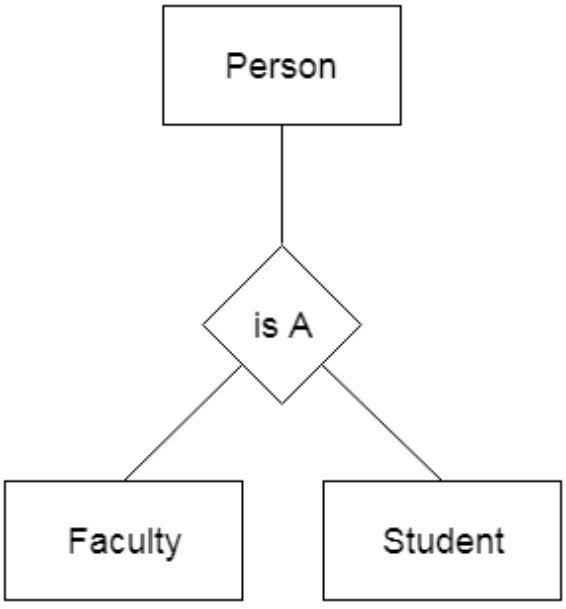
Enhanced Entity-Relationship (ER) Concepts:

In addition to the basic concepts of entities, attributes, and relationships, the Enhanced Entity-Relationship (ER) model introduces several advanced concepts to represent complex relationships and constraints. Here are three key enhanced ER concepts: Specialization/Generalization, Aggregation, and Mapping of ER model to the Relational Model.

1. Specialization/Generalization:
    - Specialization is a process of defining subclasses or specialized entities based on a base entity. It allows for the creation of distinct entities that inherit attributes and relationships from the base entity but also have their own specific attributes and relationships. Specialization is depicted using an "is-a" relationship.



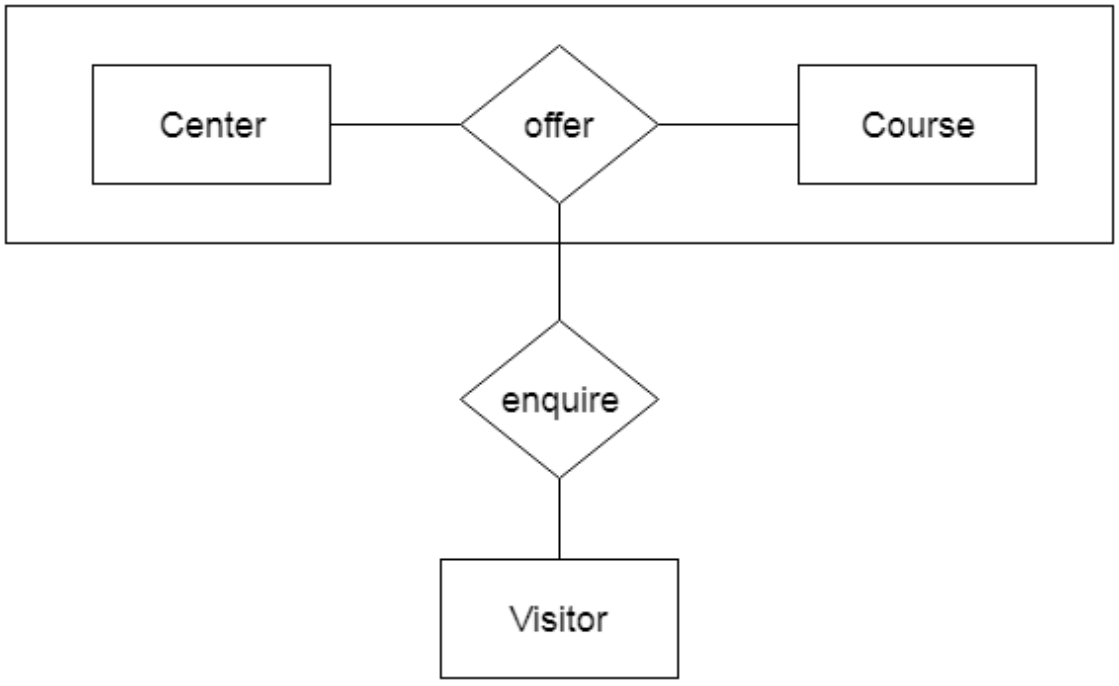
- Generalization is the reverse process, where common characteristics of several entities are abstracted to create a more generalized entity. Generalization represents a "kind-of" relationship, and the general entity encompasses the shared attributes and relationships.



2. Aggregation:

   In aggregation, the relation between two entities is treated as a single entity. In aggregation, relationship with its corresponding entities is aggregated into a higher level entity.

   For example: Center entity offers the Course entity act as a single entity in the relationship which is in a relationship with another entity visitor. In the real world, if a visitor visits a coaching center then he will never enquiry about the Course only or just about the Center instead he will ask the enquiry about both.



3. Mapping of ER Model to Relational Model:

   - The ER model is a conceptual model used to design the database structure, while the relational model is a logical model used for implementation in relational database management systems (RDBMS).
   - The mapping of the ER model to the relational model involves transforming the ER diagram into a set of relational tables.
   - Entities are mapped to tables, attributes to table columns, and relationships to foreign keys. Cardinality constraints are represented using primary keys and foreign keys to establish the relationships between tables.

- Normalization techniques are applied to ensure data integrity, minimize redundancy, and optimize the relational schema.

By incorporating specialization/generalization, aggregation, and mapping to the relational model, the Enhanced ER model provides a more expressive and flexible way to represent complex relationships and design databases that accurately reflect real-world scenarios.

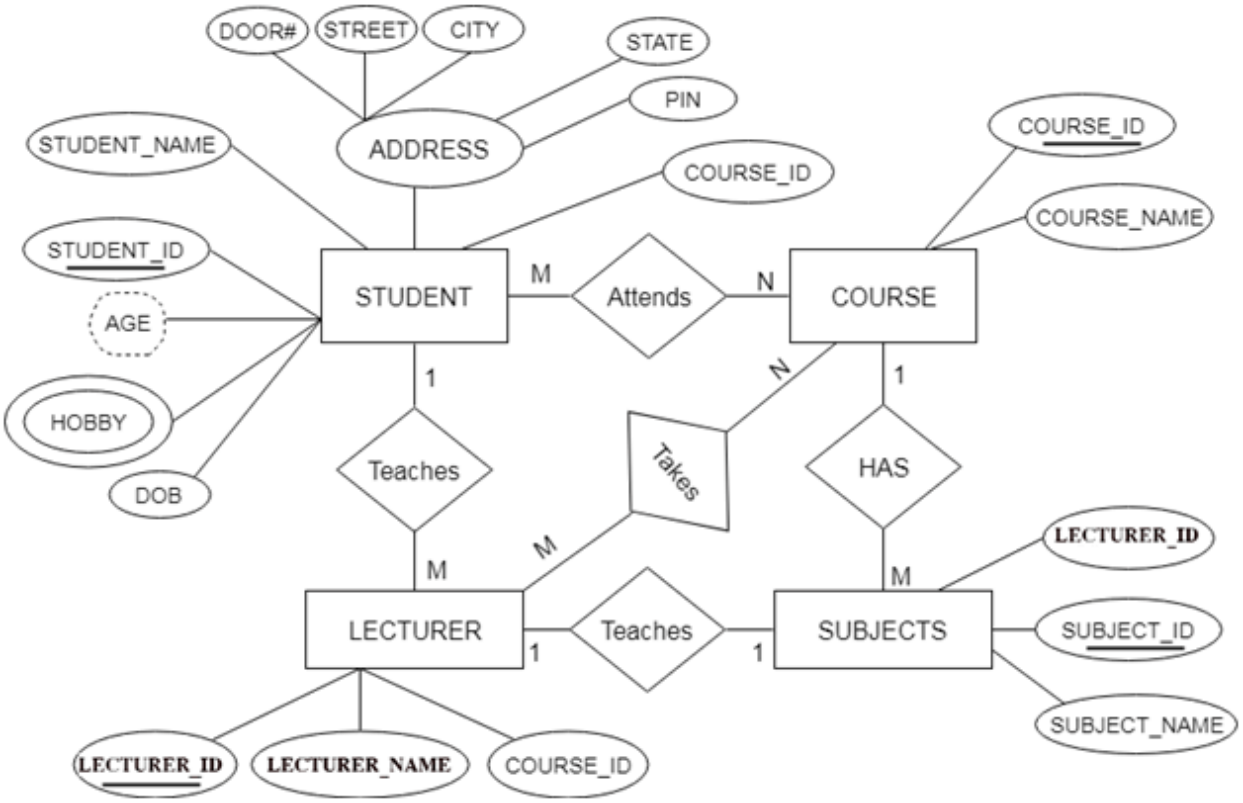
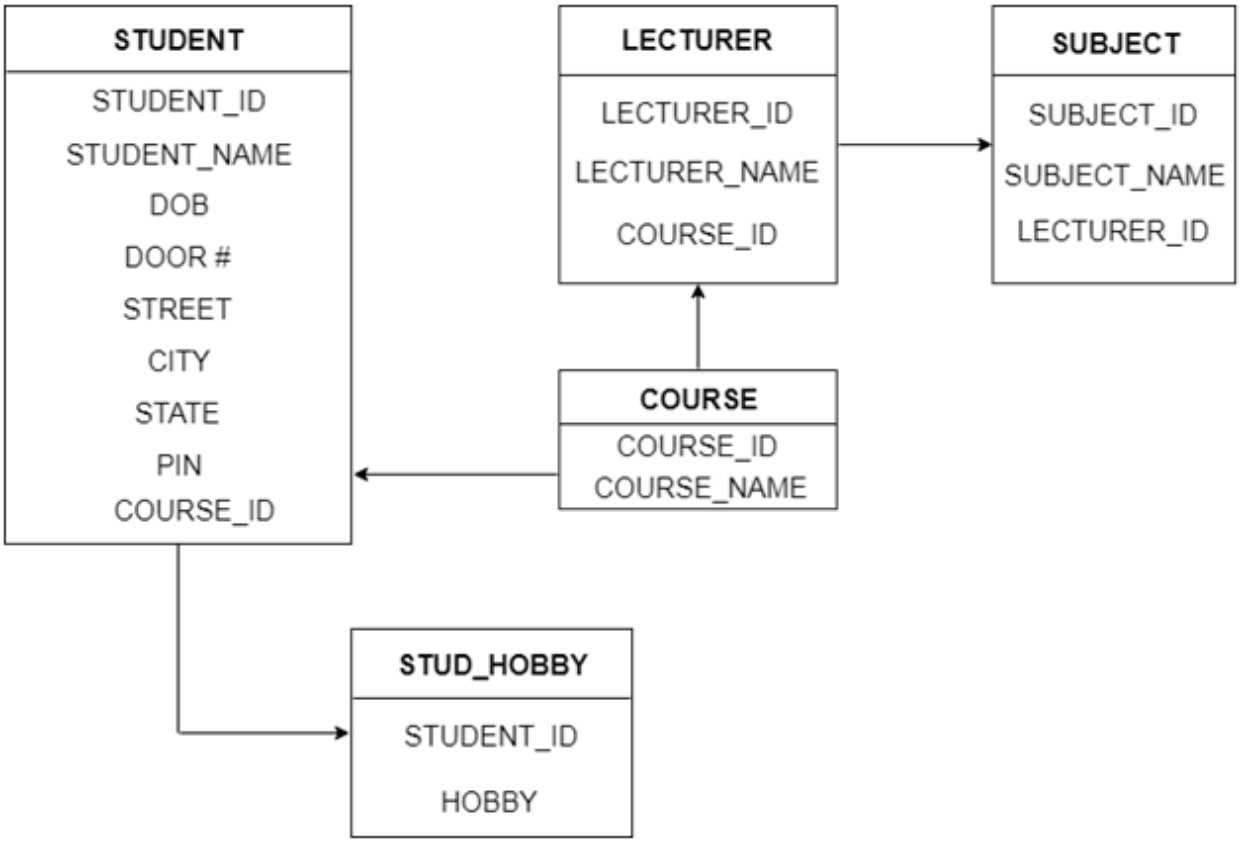


Table Structure:



SQL - DDL, DCL, and DML; Views and Indexes:

In SQL (Structured Query Language), there are different categories of statements used for various purposes. The main categories are Data Definition Language (DDL), Data Control Language (DCL), and Data Manipulation Language (DML). Additionally, SQL provides features such as views and indexes to enhance database functionality. Here's an overview of these concepts:

1. Data Definition Language (DDL):
    - DDL statements are used to define and manage the structure of the database. They allow you to create, alter, and drop database objects such as tables, views, indexes, and constraints. Some common DDL statements include:
        - CREATE: Used to create database objects like tables, views, indexes, and more.
        - ALTER: Used to modify the structure of existing database objects.
        - DROP: Used to remove database objects from the database.

2. Data Control Language (DCL):
    - DCL statements are used to control access to the database and manage user privileges. They grant or revoke permissions to users or roles for performing specific operations on database objects. Common DCL statements include:
        - GRANT: Used to give users or roles specific privileges to access and manipulate database objects.
        - REVOKE: Used to revoke previously granted privileges from users or roles.
3. Data Manipulation Language (DML):
    - DML statements are used to interact with the data stored in the database. They allow you to insert, retrieve, update, and delete data from tables. Common DML statements include:
        - SELECT: Used to retrieve data from one or more tables.
        - INSERT: Used to insert new data into a table.
        - UPDATE: Used to modify existing data in a table.
        - DELETE: Used to remove data from a table.
4. Views:
    - A view is a virtual table derived from one or more tables or other views. It is created based on a query and does not store data itself but provides a logical representation of the data.
    - Views can simplify complex queries, restrict access to certain columns or rows, or present a customized perspective of the data.
    - Views can be created using the CREATE VIEW statement and can be queried and manipulated like regular tables.
5. Indexes:
    - Indexes are data structures that improve the retrieval and searching of data in database tables. They provide a faster way to locate specific rows based on one or more columns.

- Indexes are created on specific columns of a table, and they store a sorted copy of the indexed columns' values along with a pointer to the actual data.
- Indexes speed up data retrieval operations but may slightly slow down data modification operations like inserts, updates, and deletes.
- Indexes can be created using the CREATE INDEX statement and can significantly enhance the performance of queries that involve the indexed columns.

DDL, DCL, and DML statements, along with views and indexes, form the core functionality of SQL for managing databases, defining structures, controlling access, and manipulating data efficiently.

Basics of SQL, DDL, DML, DCL, Structure, Constraints:

SQL (Structured Query Language) is a programming language used for managing relational databases. It consists of several components, including Data Definition Language (DDL), Data Manipulation Language (DML), and Data Control Language (DCL). Additionally, SQL provides various constructs for defining the structure of the database, as well as constraints to enforce data integrity. Let's explore these concepts:

1. Data Definition Language (DDL):
    - DDL is used to define and manage the structure of the database.
    - DDL statements include CREATE, ALTER, and DROP.
    - CREATE: Used to create database objects such as tables, views, indexes, and constraints.
    - ALTER: Used to modify the structure of existing database objects.
    - DROP: Used to remove database objects from the database.
2. Data Manipulation Language (DML):
    - DML is used to interact with the data stored in the database.
    - DML statements include SELECT, INSERT, UPDATE, and DELETE.
    - SELECT: Used to retrieve data from one or more tables.
    - INSERT: Used to insert new data into a table.

- UPDATE: Used to modify existing data in a table.
- DELETE: Used to remove data from a table.

3. Data Control Language (DCL):
    - DCL is used to control access to the database and manage user privileges.
    - DCL statements include GRANT and REVOKE.
    - GRANT: Used to give users or roles specific privileges to access and manipulate database objects.
    - REVOKE: Used to revoke previously granted privileges from users or roles.
4. Structure - Creation, Alteration:
    - The CREATE statement is used to create database objects such as tables, views, indexes, and constraints.
    - Example: CREATE TABLE is used to create a table with specified columns and data types.
    - The ALTER statement is used to modify the structure of existing database objects.
    - Example: ALTER TABLE is used to add or remove columns, modify column data types, or change table constraints.
5. Defining Constraints:
    - Constraints are used to enforce data integrity rules on the data stored in tables.
    - Common constraints include:
        - Primary Key: Ensures that a column or combination of columns uniquely identifies each row in a table.
        - Foreign Key: Establishes a relationship between two tables, enforcing referential integrity.
        - Unique: Ensures that a column or combination of columns have unique values.
        - Not Null: Ensures that a column does not contain null values.

    - Check: Defines a condition that must be satisfied for the data in a column.
    - IN Operator: Tests whether a value matches any value in a list of specified values.

These SQL concepts form the foundation for working with relational databases, allowing you to create structures, manipulate data, enforce constraints, and control access to the data.

UNIT - 2

Relational Model Concepts:

The relational model is a conceptual framework used in database management systems (DBMS) that organizes data into tables or relations. It consists of several key concepts that define the structure and operations of a relational database:

1. Relation/Table:
    - A relation represents a table in the relational model. It consists of rows and columns, where each row represents a record or tuple, and each column represents an attribute or field. The relation is uniquely identified by its name.
2. Tuple/Row:
    - A tuple or row represents a single record in a relation. It contains values for each attribute defined in the relation's schema.
3. Attribute/Column:
    - An attribute or column represents a specific characteristic or data field in a relation. It has a name and a data type, defining the kind of data it can store, such as text, numbers, or dates.
4. Domain:
    - A domain represents the set of allowable values for an attribute. It defines the data type and constraints associated with the attribute.

5. Primary Key:
    - A primary key is a unique identifier for each tuple in a relation. It ensures that each record in the table can be uniquely identified. Primary keys can consist of one or more attributes.
6. Foreign Key:
    - A foreign key is a reference to a primary key in another relation. It establishes relationships between tables, enforcing referential integrity.

Relational Model Constraints:

Constraints in the relational model are rules or conditions that are applied to ensure data integrity and maintain the relationships between tables. Some commonly used constraints are:

1. Primary Key Constraint:
    - Ensures that the values in the primary key column(s) of a table are unique and not null.
2. Foreign Key Constraint:
    - Ensures that values in a foreign key column(s) of a table correspond to values in the primary key column(s) of another related table.
3. Unique Constraint:
    - Ensures that the values in a column or a group of columns are unique within a table.
4. Not Null Constraint:
    - Ensures that a column does not contain null values.
5. Check Constraint:
    - Defines a condition that must be satisfied by the values in a column.

Relational Algebra:

Relational algebra is a procedural query language that operates on relations (tables) in a relational database. It provides a set of operations to retrieve and manipulate data based on specified conditions. Each operation takes one or more

relations as input and produces a new relation as output. Here are the main concepts of relational algebra:

1. Selection (σ):
    - The selection operation filters rows from a relation based on a given condition or predicate.
    - It returns a new relation that includes only the rows that satisfy the specified condition.
    - The selection operator is denoted by σ and the condition is written as a Boolean expression.
2. Projection (π):
    - The projection operation selects specific columns from a relation while eliminating duplicates.
    - It returns a new relation that includes only the specified columns.
    - The projection operator is denoted by π and the list of columns is written as a comma-separated list.
3. Union (∪):
    - The union operation combines two relations to produce a new relation that includes all tuples from both relations, removing duplicates.
    - It requires that the two relations have the same number of attributes and compatible data types.
    - The resulting relation contains all distinct tuples from both input relations.
4. Set Difference (-):
    - The set difference operation returns tuples that are present in one relation but not in the other.
    - It requires that the two relations have the same number of attributes and compatible data types.
    - The resulting relation contains tuples that exist in the first relation but not in the second.
5. Cartesian Product (×):

- The Cartesian product operation combines each tuple from one relation with every tuple from another relation.
- It returns a new relation that includes all possible combinations of tuples from both input relations.
- The resulting relation has a number of attributes equal to the sum of attributes from the input relations.

6. Join (⋈):
    - The join operation combines tuples from two or more relations based on a common attribute or set of attributes.
    - It returns a new relation that includes only the tuples that satisfy the join condition.
    - The join condition is usually defined using equality between the common attributes.
7. Intersection (∩):
    - The intersection operation returns tuples that are common to both input relations.
    - It requires that the two relations have the same number of attributes and compatible data types.
    - The resulting relation contains tuples that exist in both input relations.

Relational Calculus:

Relational calculus is a non-procedural query language that describes what data to retrieve from a relational database without specifying how to retrieve it. It provides a logical and declarative approach to defining queries by focusing on the desired results rather than the steps to obtain them. There are two types of relational calculus: Tuple Relational Calculus (TRC) and Domain Relational Calculus (DRC).

1. Tuple Relational Calculus (TRC):
    - TRC is based on the concept of selecting tuples from a relation that satisfy a given condition.
    - It uses a set of predicate formulas to specify the desired tuples in a relation.

- The predicate formulas are expressed using variables, quantifiers, and logical connectives.
- Variables represent attributes of the relation, and the quantifiers determine the scope of the variables.
- The logical connectives (AND, OR, NOT) are used to combine conditions.
- TRC queries specify the conditions that must be true for a tuple to be included in the result set.
- Example: Retrieve all customers who have placed an order: {c | ∃o (Customer(c) ∧ Order(o) ∧ c.CustomerID = o.CustomerID)}

2. Domain Relational Calculus (DRC):
    - DRC is based on the concept of selecting values from attributes that satisfy a given condition.
    - It uses a set of predicate formulas to specify the desired values in a relation's attributes.
    - The predicate formulas are expressed using variables, quantifiers, and logical connectives, similar to TRC.
    - DRC queries specify the conditions that must be true for attribute values to be included in the result set.
    - Example: Retrieve the names of all customers who have placed an order: {c.Name | ∃o (Customer(c) ∧ Order(o) ∧ c.CustomerID = o.CustomerID)}

Relational calculus allows users to define queries at a higher level of abstraction, focusing on the logical requirements of the desired data rather than the specific steps to retrieve it. The calculus expressions are translated into equivalent relational algebra expressions by query optimizers in the database management system for efficient execution.

Key concepts of relational calculus include:

- Variables: Represent attributes or values in the relations.
- Quantifiers: Specify the scope of variables, such as universal (∀) or existential (∃).

- Predicate Formulas: Express conditions using logical connectives (AND, OR, NOT) to define desired tuples or attribute values.
- Result Set: The set of tuples or attribute values that satisfy the specified conditions.

It's important to note that while relational calculus provides a theoretical foundation for query specification, it is typically not used directly by end-users. Instead, it serves as a basis for query optimization and execution within database management systems.

In relational databases, a join operation combines tuples from two or more relations based on a common attribute or set of attributes. There are several types of joins that can be used to retrieve data from multiple tables. Here are the different types of joins:

1. Inner Join:
    - An inner join returns only the matching tuples from both tables based on the join condition.
    - It selects tuples where the join attribute values are equal in both tables.
    - The resulting relation contains only the matched tuples.
    - Syntax: SELECT * FROM table1 INNER JOIN table2 ON table1.column = table2.column;
2. Left Join (or Left Outer Join):
    - A left join returns all tuples from the left table (the table specified before the JOIN keyword) and the matching tuples from the right table based on the join condition.
    - If a tuple from the left table doesn't have a matching tuple in the right table, null values are filled for the right table's attributes in the result.
    - Syntax: SELECT * FROM table1 LEFT JOIN table2 ON table1.column = table2.column;
3. Right Join (or Right Outer Join):

- A right join returns all tuples from the right table and the matching tuples from the left table based on the join condition.
- If a tuple from the right table doesn't have a matching tuple in the left table, null values are filled for the left table's attributes in the result.
- Syntax: SELECT * FROM table1 RIGHT JOIN table2 ON table1.column = table2.column;

4. Full Join (or Full Outer Join):
    - A full join returns all tuples from both tables, including the unmatched tuples.
    - If a tuple from one table doesn't have a matching tuple in the other table, null values are filled for the attributes of the respective table in the result.
    - Syntax: SELECT * FROM table1 FULL JOIN table2 ON table1.column = table2.column;
5. Cross Join (or Cartesian Join):
    - A cross join produces the Cartesian product of the two tables, meaning it combines each tuple from the first table with every tuple from the second table.
    - The resulting relation contains all possible combinations of tuples from both tables.
    - Syntax: SELECT * FROM table1 CROSS JOIN table2;
6. Self Join:
    - A self join is performed on a single table, where the table is joined with itself based on a common attribute.
    - It can be used to compare rows within the same table or establish hierarchical relationships.
    - Syntax: SELECT * FROM table1 t1 JOIN table1 t2 ON t1.column = t2.column;

SQL

In SQL, aggregate functions are used to perform calculations on a set of values and return a single value as the result. They operate on a group of rows and provide summarized information about the data. Here are some commonly used aggregate functions in SQL:

1. COUNT:
    - The COUNT function returns the number of rows in a specified column or the number of rows that match a given condition.
    - Syntax:
        - COUNT(column_name): Returns the count of non-null values in the specified column.
        - COUNT(*): Returns the count of all rows in the table, regardless of null values.
2. SUM:
    - The SUM function calculates the sum of values in a specified column.
    - It is typically used with numeric data types.
    - Syntax: SUM(column_name)
3. AVG:
    - The AVG function calculates the average (mean) value of a specified column.
    - It is typically used with numeric data types.
    - Syntax: AVG(column_name)
4. MIN:
    - The MIN function returns the minimum value in a specified column.
    - It is used to find the smallest value in a set of values.
    - Syntax: MIN(column_name)
5. MAX:
    - The MAX function returns the maximum value in a specified column.
    - It is used to find the largest value in a set of values.

- Syntax: MAX(column_name)

6. GROUP_CONCAT:
    - The GROUP_CONCAT function concatenates the values of a column into a single string, separated by a specified delimiter.
    - It is commonly used with text or string data types.
    - Syntax: GROUP_CONCAT(column_name SEPARATOR 'delimiter')

These aggregate functions can be combined with the GROUP BY clause to perform calculations on groups of data based on one or more columns. The GROUP BY clause divides the data into groups and applies the aggregate function to each group separately.

Example:

```
SELECT department, COUNT(*) as num_employees, AVG(salary) as a
vg_salary
FROM employees
GROUP BY department;
```

In the above example, the COUNT function calculates the number of employees in each department, and the AVG function calculates the average salary in each department.

Aggregate functions are powerful tools for analyzing data and generating summary information in SQL queries. They allow you to derive meaningful insights from your data by performing calculations across rows or groups of rows.

Built-in functions in SQL provide a variety of operations to manipulate and process data. Here are some commonly used categories of built-in functions:

1. Numeric Functions:
    - ABS(): Returns the absolute value of a number.
    - ROUND(): Rounds a number to a specified number of decimal places.
    - FLOOR(): Returns the largest integer less than or equal to a given number.
    - CEILING(): Returns the smallest integer greater than or equal to a given number.

- SQRT(): Calculates the square root of a number.
- POWER(): Raises a number to a specified power.

2. Date Functions:
    - NOW(): Returns the current date and time.
    - CURDATE(): Returns the current date.
    - CURTIME(): Returns the current time.
    - DATE(): Extracts the date portion from a datetime value.
    - YEAR(), MONTH(), DAY(): Extracts the year, month, or day from a date.
    - DATE_FORMAT(): Formats a date according to a specified format string.
3. String Functions:
    - CONCAT(): Concatenates two or more strings.
    - LENGTH(): Returns the length of a string.
    - SUBSTRING(): Extracts a substring from a string.
    - UPPER(), LOWER(): Converts a string to uppercase or lowercase.
    - TRIM(): Removes leading and trailing spaces from a string.
    - REPLACE(): Replaces occurrences of a substring within a string.
4. Set Operations:
    - UNION: Combines the result sets of two or more SELECT statements, removing duplicate rows.
    - UNION ALL: Combines the result sets of two or more SELECT statements, including duplicate rows.
    - INTERSECT: Returns the common rows between two result sets.
    - EXCEPT (or MINUS): Returns the rows that exist in the first result set but not in the second.
5. Subqueries:
    - A subquery is a query nested within another query, allowing you to perform complex operations.

- Subqueries can be used in various parts of a SQL statement, such as the SELECT, FROM, WHERE, and HAVING clauses.
- They are used to retrieve data based on the results of another query.
- Example: SELECT * FROM table1 WHERE column IN (SELECT column FROM table2)

These are just a few examples of the many built-in functions available in SQL. The specific functions and syntax may vary depending on the database management system you are using. It's recommended to refer to the documentation of your specific database system for a complete list of available functions and their usage.

Correlated subqueries are a type of subquery that reference one or more columns from the outer query within the subquery. In other words, the subquery's execution is dependent on the values from the outer query. Correlated subqueries are used to perform row-by-row comparisons between the outer query and the subquery. Here's an explanation of correlated subqueries and how they work:

1. Syntax:

```
SELECT column1
FROM table1 t1
WHERE condition_operator (SELECT column2 FROM table2 t2 WHE
RE t2.column = t1.column);
```

2. Execution:
   - For each row in the outer query (table1), the subquery (table2) is executed.
   - The subquery uses the value(s) from the outer query's row to filter the result.
   - The result of the subquery is then used by the outer query to determine the final result.
3. Example:
   Let's say we have two tables: "Customers" and "Orders". We want to find all customers who have placed an order.

```
SELECT CustomerName
FROM Customers c
WHERE EXISTS (SELECT 1 FROM Orders o WHERE o.CustomerID =
c.CustomerID);
```

In this example, for each customer in the outer query, the subquery checks if there exists any order in the "Orders" table with a matching CustomerID. If a match is found, the customer is included in the final result.

Correlated subqueries can be used in various parts of a SQL statement, such as the SELECT, WHERE, and HAVING clauses. They are especially useful when you need to compare values from multiple tables or perform conditional filtering based on related data.

1. GROUP BY:
    - The GROUP BY clause is used to group rows based on one or more columns.
    - It is typically used in combination with aggregate functions to calculate summaries for each group.
    - Syntax: SELECT column1, aggregate_function(column2) FROM table_name GROUP BY column1;
2. HAVING:
    - The HAVING clause is used to filter the results of a query based on a condition applied to a group.
    - It is similar to the WHERE clause but operates on the result of the GROUP BY clause.
    - Syntax: SELECT column1, aggregate_function(column2) FROM table_name GROUP BY column1 HAVING condition;
3. ORDER BY:
    - The ORDER BY clause is used to sort the result set based on one or more columns.

- It can sort in ascending (ASC) or descending (DESC) order.
- Syntax: SELECT column1, column2 FROM table_name ORDER BY column1 ASC, column2 DESC;

4. JOIN and its types:
    - JOIN is used to combine rows from two or more tables based on a related column between them.
    - The different types of JOIN include:
        - INNER JOIN: Returns only the matching rows from both tables based on the join condition.
        - LEFT JOIN (or LEFT OUTER JOIN): Returns all rows from the left table and the matching rows from the right table.
        - RIGHT JOIN (or RIGHT OUTER JOIN): Returns all rows from the right table and the matching rows from the left table.
        - FULL JOIN (or FULL OUTER JOIN): Returns all rows from both tables, including unmatched rows.
        - CROSS JOIN (or Cartesian Join): Returns the Cartesian product of rows from both tables.
5. EXISTS, ANY, ALL:
    - EXISTS is a logical operator used to check the existence of a subquery result.
    - ANY and ALL are comparison operators used in combination with a subquery.
    - EXISTS: Returns true if the subquery returns any rows, otherwise false.
    - ANY: Returns true if the comparison condition is true for any value in the subquery result.
    - ALL: Returns true if the comparison condition is true for all values in the subquery result.
6. View and its types:
    - A view is a virtual table derived from the result of a query.

- It is a saved SQL query that can be treated as a table in subsequent queries.
- Views provide a way to simplify complex queries, restrict data access, and present a logical representation of data.
- Types of views include:
    - Simple View: A view created from a single table or multiple tables with a straightforward query.
    - Complex View: A view created using joins, subqueries, or other complex SQL constructs.
    - Materialized View: A view that stores the result of the query physically, providing faster access to the data.

Views can be used to enhance security, encapsulate complex logic, and provide a simplified interface to the underlying data.

Transactions

A transaction is a unit of work that is performed against a database. Transactions are units or sequences of work accomplished in a logical order, whether in a manual fashion by a user or automatically by some sort of a database program.

A transaction is the propagation of one or more changes to the database. For example, if you are creating a record or updating a record or deleting a record from the table, then you are performing a transaction on that table. It is important to control these transactions to ensure the data integrity and to handle database errors.

Practically, you will club many SQL queries into a group and you will execute all of them together as a part of a transaction.

Properties of Transactions

Transactions have the following four standard properties, usually referred to by the acronym **ACID**.

- **Atomicity** – ensures that all operations within the work unit are completed successfully. Otherwise, the transaction is aborted at the point of failure and all the previous operations are rolled back to their former state.
- **Consistency** – ensures that the database properly changes states upon a successfully committed transaction.
- **Isolation** – enables transactions to operate independently of and transparent to each other.
- **Durability** – ensures that the result or effect of a committed transaction persists in case of a system failure.

Transaction Control

The following commands are used to control transactions.

- **COMMIT** – to save the changes.
- **ROLLBACK** – to roll back the changes.
- **SAVEPOINT** – creates points within the groups of transactions in which to ROLLBACK.
- **SET TRANSACTION** – Places a name on a transaction.

AD

Transactional Control Commands

Transactional control commands are only used with the **DML Commands** such as - INSERT, UPDATE and DELETE only. They cannot be used while creating tables or dropping them because these operations are automatically committed in the database.

The COMMIT Command

The COMMIT command is the transactional command used to save changes invoked by a transaction to the database.

The COMMIT command is the transactional command used to save changes invoked by a transaction to the database. The COMMIT command saves all the transactions to the database since the last COMMIT or ROLLBACK command.

The syntax for the COMMIT command is as follows.

```
COMMIT;
```

Example

Consider the CUSTOMERS table having the following records –

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

Following is an example which would delete those records from the table which have age = 25 and then COMMIT the changes in the database.

```
SQL> DELETE FROM CUSTOMERS
   WHERE AGE = 25;
SQL> COMMIT;
```

Output

Thus, two rows from the table would be deleted and the SELECT statement would produce the following result.

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
```

```
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

The ROLLBACK Command

The ROLLBACK command is the transactional command used to undo transactions that have not already been saved to the database. This command can only be used to undo transactions since the last COMMIT or ROLLBACK command was issued.

The syntax for a ROLLBACK command is as follows –

```
ROLLBACK;
```

Example

Consider the CUSTOMERS table having the following records –

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

Following is an example, which would delete those records from the table which have the age = 25 and then ROLLBACK the changes in the database.

```
SQL> DELETE FROM CUSTOMERS
   WHERE AGE = 25;
SQL> ROLLBACK;
```

Output

Thus, the delete operation would not impact the table and the SELECT statement would produce the following result.

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

The SAVEPOINT Command

A SAVEPOINT is a point in a transaction when you can roll the transaction back to a certain point without rolling back the entire transaction.

The syntax for a SAVEPOINT command is as shown below.

```
SAVEPOINT SAVEPOINT_NAME;
```

This command serves only in the creation of a SAVEPOINT among all the transactional statements. The ROLLBACK command is used to undo a group of transactions.

The syntax for rolling back to a SAVEPOINT is as shown below.

```
ROLLBACK TO SAVEPOINT_NAME;
```

Following is an example where you plan to delete the three different records from the CUSTOMERS table. You want to create a SAVEPOINT before each delete, so that you can ROLLBACK to any SAVEPOINT at any time to return the appropriate data to its original state.

Example

Consider the CUSTOMERS table having the following records.

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

The following code block contains the series of operations.

```
SQL> SAVEPOINT SP1;
Savepoint created.
SQL> DELETE FROM CUSTOMERS WHERE ID=1;
1 row deleted.
SQL> SAVEPOINT SP2;
Savepoint created.
SQL> DELETE FROM CUSTOMERS WHERE ID=2;
1 row deleted.
SQL> SAVEPOINT SP3;
Savepoint created.
SQL> DELETE FROM CUSTOMERS WHERE ID=3;
```

```
1 row deleted.
```

Now that the three deletions have taken place, let us assume that you have changed your mind and decided to ROLLBACK to the SAVEPOINT that you identified as SP2. Because SP2 was created after the first deletion, the last two deletions are undone –

```
SQL> ROLLBACK TO SP2;
Rollback complete.
```

Output

Notice that only the first deletion took place since you rolled back to SP2.

```
SQL> SELECT * FROM CUSTOMERS;
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
6 rows selected.
```

The RELEASE SAVEPOINT Command

The RELEASE SAVEPOINT command is used to remove a SAVEPOINT that you have created.

The syntax for a RELEASE SAVEPOINT command is as follows.

```
RELEASE SAVEPOINT SAVEPOINT_NAME;
```

Once a SAVEPOINT has been released, you can no longer use the ROLLBACK command to undo transactions performed since the last SAVEPOINT.

The SET TRANSACTION Command

The SET TRANSACTION command can be used to initiate a database transaction. This command is used to specify characteristics for the transaction that follows. For example, you can specify a transaction to be read only or read write.

Syntax

The syntax for a SET TRANSACTION command is as follows.

```
SET TRANSACTION [ READ WRITE | READ ONLY ];
```

SQL Example

Tables:

1. Customers
    - Columns: CustomerID (Primary Key), CustomerName, City, Country
2. Orders
    - Columns: OrderID (Primary Key), CustomerID (Foreign Key), OrderDate, TotalAmount

Now, let's run SQL queries on different topics:

Aggregate Functions - SUM, AVG, COUNT:

```
SELECT SUM(TotalAmount) AS TotalSum, AVG(TotalAmount) AS Avera
geAmount, COUNT(*) AS TotalOrders FROM Orders;
```

Built-in Functions - Numeric, Date, String:

```
SELECT ABS(-10) AS AbsoluteValue, ROUND(7.86) AS RoundedValue,
DATE_FORMAT(NOW(), '%Y-%m-%d') AS CurrentDate, CONCAT('Hello',
' ', 'World') AS ConcatenatedString;
```

Set Operations - UNION, INTERSECT, EXCEPT (MINUS):

```
SELECT CustomerName FROM Customers WHERE Country = 'USA' UNION
SELECT CustomerName FROM Customers WHERE Country = 'Canada';
```

Subqueries:

```
SELECT CustomerName, (SELECT COUNT(*) FROM Orders WHERE Order
s.CustomerID = Customers.CustomerID) AS OrderCount FROM Custom
ers;
```

Correlated Subqueries:

```
SELECT CustomerName FROM Customers c WHERE EXISTS (SELECT 1 FR
OM Orders o WHERE o.CustomerID = c.CustomerID);
```

GROUP BY and HAVING:

```
SELECT CustomerID, COUNT(*) AS OrderCount FROM Orders GROUP BY
CustomerID HAVING OrderCount > 2;
```

ORDER BY:

```
SELECT CustomerName, TotalAmount FROM Customers c JOIN Orders
o ON c.CustomerID = o.CustomerID ORDER BY TotalAmount DESC;
```

JOIN - INNER JOIN, LEFT JOIN:

```
SELECT Customers.CustomerName, Orders.OrderID FROM Customers I
NNER JOIN Orders ON Customers.CustomerID = Orders.CustomerID;
```

EXISTS, ANY, ALL:

```
SELECT CustomerName FROM Customers WHERE CustomerID = ANY (SEL
ECT CustomerID FROM Orders);
```

View - Simple View:

```
CREATE VIEW CustomerView AS SELECT CustomerID, CustomerName, C
ity, Country FROM Customers;
```

These queries are written in SQL syntax and properly formatted in code boxes for better readability. Make sure to adjust the table and column names according to your actual database schema.

Unit 3

Best resource to learn everything about database normalization:

https://www.youtube.com/watch?v=MjwaP18s0Xs&list=PLdo5W4Nhv31b33kF46f9aFjoJPOkdlsRc&index=3&pp=iAQB

Functional Dependencies in Relational Databases:

Functional dependencies describe the relationship between attributes in a relational database. They represent the dependencies of one set of attributes (known as the determinant) on another set of attributes (known as the dependent). The notation used to represent functional dependencies is X → Y, where X determines Y.

Here are some key concepts related to functional dependencies:

1. Determinant: The attribute or set of attributes that uniquely determines the value of another attribute(s).
2. Dependent: The attribute(s) whose value is functionally dependent on the determinant.

3. Fully Functional Dependency: A functional dependency where removing any attribute(s) from the determinant would break the dependency.
4. Partial Functional Dependency: A functional dependency where removing some attribute(s) from the determinant would not break the dependency.

Normalization for Relational Databases:

Normalization is the process of organizing data in a relational database to reduce redundancy, dependency, and anomalies. It involves applying a series of normal forms (such as 1NF, 2NF, 3NF, etc.) to improve data integrity and maintainability. Here are the commonly used normal forms:

1. **First Normal Form (1NF):**
    - Eliminate duplicate rows by ensuring each attribute contains a single value.
    - Create separate tables for related data.
    - Identify a primary key for each table.
2. **Second Normal Form (2NF):**
    - Meet the requirements of 1NF.
    - Remove partial dependencies (attributes depend on the entire primary key, not just a part of it).
    - Split the table into multiple tables if necessary.
3. **Third Normal Form (3NF):**
    - Meet the requirements of 2NF.
    - Eliminate transitive dependencies (attributes depend only on the primary key, not on other non-key attributes).
    - Split the table into multiple tables if necessary.
4. **Boyce-Codd Normal Form (BCNF):**
    - Meet the requirements of 3NF.
    - Eliminate any remaining dependencies (functional dependencies on candidate keys).
    - Split the table into multiple tables if necessary.

5. **Fourth Normal Form (4NF) and higher forms:**
    - Address multi-valued dependencies and other complex relationships.
    - Achieve further normalization if needed.

Normalization helps in reducing data redundancy, improving data integrity, simplifying updates, and increasing the overall efficiency of the database.

It's important to note that normalization is an iterative process, and higher normal forms may not always be necessary depending on the specific requirements and complexity of the database schema.

Lossless Join and Dependency Preserving Decomposition in Database Normalization:

When decomposing a relation into multiple smaller relations during the normalization process, two important concepts to consider are lossless join and dependency preservation. Let's explore each concept in detail:

1. Lossless Join:
    - Lossless join ensures that when the decomposed relations are rejoined through a common attribute or set of attributes, the original relation can be reconstructed without any loss of information.
    - It guarantees that all the original tuples (rows) can be recreated by joining the decomposed relations using the appropriate join operation.
    - Lossless join is important because it ensures that the decomposed relations maintain the same information as the original relation.
    - To achieve lossless join, the decomposition must be based on a set of attributes that forms a superkey for at least one of the decomposed relations.
    - If a decomposition does not have the lossless join property, it may result in the creation of additional rows during the join operation, leading to data loss or incorrect results.
2. Dependency Preservation:

- Dependency preservation ensures that the functional dependencies that held in the original relation also hold in the decomposed relations.
- It means that any dependency implied by the original set of functional dependencies should still be enforced after the decomposition.
- Preserving dependencies is crucial to maintain data integrity and consistency in the decomposed relations.
- If a decomposition loses any of the original functional dependencies, it may lead to incorrect results and data inconsistencies.
- Dependency preservation is typically achieved by carefully selecting the decomposition scheme and ensuring that the decomposed relations capture all the relevant dependencies.

During the normalization process, it is desirable to achieve both lossless join and dependency preservation in the decomposed relations. However, it is not always possible to achieve both simultaneously. In such cases, a trade-off must be made to prioritize one over the other, depending on the specific requirements and constraints of the database design.

In addition to the commonly known normal forms (1NF, 2NF, 3NF, BCNF), there are additional normal forms that address specific types of dependencies and further refine the database design. These include:

1. Fourth Normal Form (4NF):
    - Fourth Normal Form (4NF) deals with multivalued dependencies (MVDs) within a relation.
    - A multivalued dependency occurs when a relationship between two sets of attributes creates a situation where one set of attributes determines multiple independent sets of values for another set of attributes.
    - To achieve 4NF, the relation must be in 3NF, and any non-key attribute must be functionally dependent on the entire primary key.
    - If a relation contains non-key attributes that are independent of each other, they should be separated into their own relation.
2. Fifth Normal Form (5NF) or Project-Join Normal Form (PJNF):
    - Fifth Normal Form (5NF) deals with join dependencies in a relation.

- A join dependency occurs when a relation can be logically reconstructed by joining two or more smaller relations.
- To achieve 5NF, the relation must be in 4NF, and all join dependencies must be explicitly represented.
- If a relation has join dependencies that are not already expressed, additional relations need to be created to represent these dependencies.

3. Domain-Key Normal Form (DK/NF):
    - Domain-Key Normal Form (DK/NF) is a less commonly known normal form that addresses dependencies involving keys and non-key attributes within a relation.
    - It ensures that every constraint on a relation is a logical consequence of the domain constraints and the key constraint.
    - A relation in DK/NF is already in higher normal forms (such as 3NF or BCNF) and does not contain any spurious tuples.
    - Achieving DK/NF helps ensure that the relation is free from any unnecessary redundancy and dependency violations.

It's important to note that 4NF, 5NF, and DK/NF are considered advanced normal forms and may not always be necessary for every database design. The application of these normal forms depends on the specific requirements, complexity, and dependencies present in the database schema.

Transaction schedule

A transaction schedule, also known as a transaction execution schedule or simply a schedule, is a chronological order of the execution of transactions in a database system. It represents the sequence of operations performed by different transactions over time. A transaction schedule can be visualized as a table or a timeline.

Components of a Transaction Schedule:

1. Transactions: The schedule includes multiple transactions, each denoted by a unique identifier. Transactions are represented by T1, T2, T3, and so on.
2. Operations: Each transaction in the schedule consists of a series of operations such as reads (R) and writes (W) performed on database objects. For example,

T1 may read a value from an object X (denoted as R(X)) and then write a value to object Y (denoted as W(Y)).

3. Timestamps: Transactions in the schedule are typically associated with timestamps to indicate the order in which they execute. The timestamps can be actual time values or logical values assigned by the system.

Properties of Transaction Schedules:

1. Serial Schedule: A serial schedule is a schedule where transactions are executed one after another without any overlap. In a serial schedule, the operations of each transaction are executed in their entirety before moving to the next transaction.
2. Concurrent Schedule: A concurrent schedule is a schedule where multiple transactions are executed simultaneously or in an interleaved manner. Concurrent schedules offer better performance by allowing parallel execution of transactions.
3. Serializability: Serializability is a property of a schedule that ensures that its execution is equivalent to some serial execution of the same transactions. In other words, the final state of the database remains the same as if the transactions were executed in a serialized manner.
4. Conflict Serializable Schedule: A conflict serializable schedule is a concurrent schedule that produces the same result as some serial schedule. It ensures that conflicting operations (e.g., read-write or write-write conflicts) between transactions are properly ordered to maintain data consistency.
5. Recoverability: Recoverability refers to the ability to recover the database to a consistent state in the event of a failure. A schedule is recoverable if, for every transaction T that reads a value written by another transaction S, the commit operation of S appears before the read operation of T.

Transaction schedules play a crucial role in ensuring the correctness, consistency, and concurrency control of database operations. They provide a systematic representation of transaction execution, enabling efficient and reliable data management.

Serializability

Serializability is a property of transaction schedules in a database system. It ensures that the concurrent execution of multiple transactions produces the same results as if the transactions were executed in a serialized (sequential) manner. In other words, serializability guarantees that the final state of the database remains consistent and correct.

Concurrency Control Techniques:
Concurrency control techniques are mechanisms employed to manage the simultaneous execution of multiple transactions while preserving data consistency and avoiding conflicts. These techniques aim to ensure serializability and prevent issues like data inconsistency, lost updates, and dirty reads. Here are some commonly used concurrency control techniques:

1. Lock-Based Concurrency Control:
    - Lock-based techniques involve the use of locks to control access to shared resources (e.g., database objects) during transaction execution.
    - Two popular lock-based protocols are:
        - Two-Phase Locking (2PL): Transactions acquire locks (e.g., shared locks, exclusive locks) on objects before accessing them and release the locks only after the transaction is completed or committed.
        - Optimistic Concurrency Control (OCC): Transactions proceed without acquiring locks initially, but they perform validation before committing to check for any conflicts with other transactions.
2. Multiversion Concurrency Control (MVCC):
    - MVCC allows multiple versions of a database object to exist simultaneously to provide concurrent access.
    - Each transaction works with a specific snapshot of the database, and changes made by one transaction do not affect the snapshots of other concurrent transactions.
    - MVCC eliminates the need for exclusive locks and provides higher concurrency.
3. Timestamp Ordering:

- Timestamp ordering assigns unique timestamps to each transaction based on their start times.
- Read and write operations are ordered based on the transaction timestamps to ensure consistency and avoid conflicts.
- Transactions with earlier timestamps are given priority over those with later timestamps.

4. Conflict Serializability:
    - Conflict serializability is a technique that ensures that a transaction schedule is conflict equivalent to some serial execution.
    - Conflict equivalence means that all conflicting pairs of operations in the schedule have the same order as in the corresponding serial execution.

Locking Techniques:
Locking is a commonly used mechanism in concurrency control. Different types of locks can be employed to control access to shared resources. Some important locking techniques include:

1. Shared Lock (S-Lock):
    - Allows concurrent read access to a shared resource by multiple transactions.
    - Multiple transactions can hold shared locks on the same resource simultaneously.
2. Exclusive Lock (X-Lock):
    - Allows exclusive write access to a resource, preventing any other transaction from accessing or modifying it.
    - Only one transaction can hold an exclusive lock on a resource at a time.
3. Intent Lock:
    - Indicates the intent of a transaction to acquire shared or exclusive locks on multiple resources.
    - Helps in coordinating lock acquisition and preventing conflicts.
4. Deadlock Detection and Avoidance:

- Deadlock detection and avoidance techniques are employed to identify and resolve situations where transactions are waiting for each other indefinitely.
- Techniques like deadlock detection algorithms (e.g., resource allocation graph) and deadlock prevention strategies (e.g., deadlock avoidance by resource ordering) can be used.

Concurrency control and locking techniques are crucial for maintaining data consistency and allowing efficient parallel execution of transactions in a multi-user database environment.

Timestamp Ordering:

Timestamp ordering is a concurrency control technique that assigns unique timestamps to each transaction based on their start times. It ensures that the execution of transactions is properly ordered to maintain data consistency and avoid conflicts. Here's how timestamp ordering works:

1. Assigning Timestamps:
    - When a transaction starts, it is assigned a unique timestamp, either based on the system time or a logical value provided by the database system.
    - The timestamp represents the relative order of the transaction's start time compared to other transactions.
2. Read and Write Operations:
    - During transaction execution, read and write operations are ordered based on the timestamps assigned to the transactions.
    - A read operation of a transaction T can only access a value produced by another transaction S if the timestamp of S is less than the timestamp of T.
    - Similarly, a write operation of a transaction T can only modify a value previously accessed by another transaction S if the timestamp of S is less than the timestamp of T.
3. Conflict Detection:
    - A conflict occurs when two transactions access the same data item and at least one of them performs a write operation.

- In timestamp ordering, conflicts are detected by comparing the timestamps of conflicting operations.
- If the timestamps satisfy the ordering rules (T1 < T2), there is no conflict. Otherwise, a conflict exists.

4. Handling Conflicts:
    - Conflicts can be resolved using various techniques, such as:
        - Abort: If a conflict is detected, one of the transactions can be aborted and rolled back to its previous state. The aborted transaction can be retried later.
        - Wait: Alternatively, the transaction with the lower timestamp can be made to wait until the conflicting transaction completes its operation.

Recoverable Schedules:

Recoverable schedules ensure that a transaction can be rolled back or recovered to a consistent state in the event of a failure or an aborted transaction. A schedule is considered recoverable if, for every transaction T that reads a value written by another transaction S, the commit operation of S appears before the read operation of T. Here's an explanation:

1. Transaction Dependencies:
    - In a schedule, if transaction T1 reads a value that was written by transaction T2, T1 has a dependency on T2.
    - The dependency is established to ensure that T1 reads a consistent and valid value from T2.
2. Recoverability Condition:
    - A schedule is recoverable if, for every transaction T1 that reads a value written by another transaction T2, the commit operation of T2 appears before the read operation of T1.
3. Ensuring Recoverability:
    - If a schedule violates recoverability, where T1 reads a value written by T2 before T2 commits, it can lead to inconsistencies if T2 later aborts.

- To ensure recoverability, the schedule can be modified by adding appropriate wait operations to ensure that T1 reads only committed values.

By maintaining recoverable schedules, the database system can ensure that the effects of committed transactions are durable and consistent, even in the presence of failures or aborted transactions.

Granularity of a data item

Granularity of a data item refers to the level of detail or the size of the data unit being considered or manipulated. It determines the extent to which data is divided or aggregated. In the context of a database system, granularity is an important concept in data management and represents the level at which data is stored, accessed, or processed. It can vary depending on the specific requirements of the application or the nature of the data being dealt with.

Here are different levels of granularity for data items:

1. Fine-Grained Granularity:
    - Fine-grained granularity refers to a smaller or more detailed level of data representation or processing.
    - Data items at this level are often individual attributes or fields within a record or entity.
    - Fine-grained granularity allows for more precise control and manipulation of data but may require more storage and processing resources.
2. Coarse-Grained Granularity:
    - Coarse-grained granularity represents a higher or more aggregated level of data.
    - Data items at this level are larger units, such as entire records, entities, or collections of related attributes.
    - Coarse-grained granularity can improve performance by reducing the number of individual data items to be processed but may sacrifice precision or fine-grained control.

The choice of granularity depends on the specific requirements and characteristics of the application and the operations being performed on the data. Different levels of granularity may be used for different purposes within a database system.

Examples of granularity in different contexts:

1. Storage Granularity:
    - In storage systems, granularity refers to the size of the storage unit, such as a disk block or a page.
    - Fine-grained granularity would involve smaller storage units, allowing for more precise allocation and management of data.
    - Coarse-grained granularity would involve larger storage units, providing better performance in terms of reading and writing larger chunks of data.
2. Locking Granularity:
    - In concurrency control, granularity relates to the level at which locks are acquired and released.
    - Fine-grained locking involves acquiring locks on individual data items, such as fields or records, allowing for more concurrent access.
    - Coarse-grained locking involves acquiring locks on larger units, reducing concurrency but simplifying management and reducing overhead.
3. Query Granularity:
    - In database queries, granularity relates to the level at which data is accessed or aggregated.
    - Fine-grained queries retrieve and manipulate individual data items or small subsets of data.
    - Coarse-grained queries involve aggregating data at higher levels, such as summarizing data across multiple entities or groups.

Determining the appropriate granularity for data items is a design decision that should be based on factors like performance requirements, data dependencies, concurrency considerations, and the nature of the application and data model.

Deadlock Detection and Recovery:

Deadlock is a situation in a database system where two or more transactions are waiting indefinitely for each other to release resources, resulting in a circular

dependency. Deadlock detection and recovery mechanisms are essential to handle such situations and ensure the system can continue functioning properly. Here are the key concepts related to deadlock detection and recovery:

1. Deadlock Detection:
    - Deadlock detection is the process of identifying the existence of a deadlock in the system.
    - Various algorithms, such as the resource allocation graph algorithm, can be used to detect deadlocks by analyzing the resource dependencies among transactions.
2. Resource Allocation Graph Algorithm:
    - The resource allocation graph algorithm represents the resource allocation and wait-for relationships between transactions and resources.
    - It identifies deadlocks by detecting cycles in the graph.
    - If a cycle is found, it indicates the presence of a deadlock.
3. Deadlock Recovery:
    - Deadlock recovery involves taking actions to break the deadlock and allow the system to continue functioning.
    - There are two common approaches to deadlock recovery:
        - Deadlock Detection and Abort: Once a deadlock is detected, one or more transactions involved in the deadlock can be aborted to break the circular dependency and release the resources.
        - Deadlock Detection and Resource Preemption: Instead of aborting transactions, the system can selectively preempt resources from transactions to resolve the deadlock. The preemption can be based on certain predefined rules or priorities.

Recovery Techniques:

Recovery techniques in database systems ensure data consistency and availability in the event of failures or catastrophic events. These techniques involve making backups of the database and employing mechanisms to restore the database to a consistent state. Here are the key concepts related to recovery techniques:

1. Recovery Concepts:
    - Recovery refers to the process of restoring the database to a correct and consistent state after a failure or error.
    - The recovery process typically involves two phases: undo and redo.
        - Undo Phase: In this phase, the system rolls back or undoes the effects of incomplete or aborted transactions to restore the database to its previous consistent state.
        - Redo Phase: In this phase, the system applies changes made by committed transactions that were not yet reflected in the database due to failure or interruption.
2. Database Backup:
    - Database backup involves creating copies of the database at a specific point in time to provide a restore point in case of data loss or corruption.
    - Different backup strategies can be employed, such as full backups (copying the entire database), incremental backups (copying only the changes since the last backup), or differential backups (copying the changes since the last full backup).
3. Recovery from Catastrophic Failures:
    - Catastrophic failures refer to severe incidents like hardware failures, natural disasters, or system crashes that can lead to data loss or damage.
    - Recovery from catastrophic failures involves restoring the database using the most recent backup and applying the redo logs or transaction logs to bring the database up to date.
    - It is important to have off-site backups or remote replication to ensure data availability in case of catastrophic failures at the primary location.

Database backup and recovery are crucial for data protection and continuity. Regular backups, well-defined recovery procedures, and the use of redundant and distributed systems are essential for ensuring data resilience and minimizing downtime.

Control Structures:

Control structures allow you to control the flow of execution and perform conditional or iterative operations in your database programming. Let's look at some examples:

- IF-ELSE statement:

```
IF condition THEN
    -- Code block executed when the condition is true
    -- Perform desired actions
ELSE
    -- Code block executed when the condition is false
    -- Perform alternative actions
END IF;
```

- WHILE loop:

```
WHILE condition DO
    -- Code block executed repeatedly as long as the condition
is true
    -- Perform desired actions
END WHILE;
```

- FOR loop (example in PL/SQL):

```
FOR counter IN 1..10 LOOP
    -- Code block executed for each iteration of the loop
    -- Perform desired actions
END LOOP;
```

Exception Handling

Exception handling is used to catch and handle errors or exceptional conditions that may occur during the execution of database programs. Here are some examples:

- TRY-CATCH block (example in PL/SQL):

```
BEGIN
    -- Code block where exceptions might occur
    -- Perform desired actions
EXCEPTION
    WHEN exception_type THEN
        -- Code block executed when a specific exception occur
s
        -- Handle the exception or perform error handling acti
ons
    WHEN OTHERS THEN
        -- Code block executed when any other exception occurs
        -- Handle the exception or perform error handling acti
ons
END;
```

- Exception propagation:

```
DECLARE
    -- Declare variables and statements
BEGIN
    -- Call a procedure or execute a statement that may raise
an exception
    -- The exception can be propagated up the call stack to be
caught and handled in an outer block
EXCEPTION
    WHEN exception_type THEN
        -- Handle the exception or perform error handling acti
ons
END;
```

Stored Procedures:

Stored procedures are pre-compiled sets of SQL statements that are stored and

executed on the database server. They allow you to encapsulate and reuse commonly performed operations. Here's an example:

```
-- Example of a stored procedure in SQL Server
CREATE PROCEDURE GetCustomerDetails
    @customerId INT
AS
BEGIN
    -- SQL statements to retrieve customer details based on th
e provided ID
    SELECT * FROM Customers WHERE CustomerID = @customerId;
END;
```

Triggers

Triggers are database objects that are automatically executed or fired in response to specific events or actions occurring on a table. They can be used to enforce business rules, maintain data integrity, or perform additional actions. Here's an example:

```
-- Example of an AFTER INSERT trigger in MySQL
CREATE TRIGGER NewOrderTrigger
AFTER INSERT ON Orders
FOR EACH ROW
BEGIN
    -- Code executed after a new row is inserted into the Orde
rs table
    -- Perform desired actions, such as updating related table
s or generating notifications
END;
```

In this example, the trigger "NewOrderTrigger" is fired after an insert operation is performed on the "Orders" table, allowing you to perform additional actions based on the newly inserted data.

These examples illustrate how control structures, exception handling, stored procedures, and triggers are used in database programming. Remember that the specific syntax and features may vary depending on the database management system you are using.

Unit 4

Secondary Storage Devices

Secondary storage devices, also known as auxiliary or external storage devices, are used to store data persistently outside the primary memory (RAM). They provide long-term storage for large amounts of data even when the power is turned off. Some common types of secondary storage devices include hard disk drives (HDDs), solid-state drives (SSDs), optical disks (CDs, DVDs, Blu-ray), and magnetic tapes.

Operations on Files:
Files are logical units of data storage that represent collections of related records. Various operations can be performed on files to manage and manipulate data. Some common operations on files include:

1. Creating a file: Allocating space on a storage device to store a new file and defining its attributes.
2. Opening a file: Associating a file handle or pointer with an existing file to access its contents.
3. Reading from a file: Retrieving data from a file into the memory for processing or display.
4. Writing to a file: Storing data from the memory into a file for persistent storage.
5. Appending to a file: Adding new data at the end of an existing file without overwriting the existing content.
6. Updating a file: Modifying the existing data in a file by replacing or modifying specific records.

7. Deleting a file: Removing a file from the storage device, freeing up the allocated space.

Heap Files

Heap files are a basic file organization technique where records are stored in no particular order. When a new record is inserted, it is placed at the end of the file. As a result, records can be scattered throughout the file, and there is no specific order or indexing mechanism. Retrieving data from a heap file requires scanning the entire file sequentially.

Operations on heap files include:

1. Insertion: New records are added at the end of the file.
2. Reading: Scanning the entire file sequentially to find the desired records.
3. Updating: Modifying the data of specific records.
4. Deletion: Removing specific records from the file, leaving gaps.

Heap files are simple to implement but can suffer from poor performance when dealing with large amounts of data, as every operation requires a full scan of the file.

Sorted Files:

Sorted files are a file organization technique where records are stored in a specified order based on the values of one or more fields. The primary advantage of sorted files is that they allow for efficient searching and retrieval of data using techniques like binary search or index-based access.

Operations on sorted files include:

1. Insertion: New records are inserted at the correct position in the sorted order.
2. Reading: Efficient searching and retrieval of records based on the sorted order.
3. Updating: Modifying the data of specific records while maintaining the sorted order.
4. Deletion: Removing specific records from the file while maintaining the sorted order.

Sorted files are beneficial for applications that require frequent searching and retrieval based on the sorted field(s). However, maintaining the sorted order during insertions and deletions can be time-consuming and resource-intensive.

It's important to choose the appropriate file organization technique based on the specific requirements of your application, considering factors such as data access patterns, efficiency, and scalability.

Hashing

Hashing is a technique used in computer science and databases to efficiently map data to a fixed-size array or hash table. It involves applying a hash function to a data item to generate a hash value, which is then used as an index to store or retrieve the data from the hash table.

Here's how the hashing process works:

1. Hash Function: A hash function takes an input (data item) and computes a hash value, which is a fixed-size output. The hash function should have the following properties:
    - Deterministic: For the same input, the hash function should always produce the same hash value.
    - Uniform Distribution: The hash function should distribute the hash values uniformly across the hash table to minimize collisions.
2. Hash Table: A hash table is an array of fixed size that stores data items based on their hash values. Each slot or bucket in the hash table corresponds to an index, which is determined by applying the hash function to the data item.
3. Insertion:
    - The data item is passed through the hash function to generate the hash value.
    - The hash value is used as an index to store the data item in the hash table. If there is already data stored at that index (collision), one of the collision resolution techniques is applied (e.g., chaining or open addressing) to handle the collision.
4. Retrieval:

- The data item to be retrieved is passed through the hash function to generate the hash value.
- The hash value is used as an index to locate the data item in the hash table. If there was a collision during insertion, the collision resolution technique is used to locate the correct data item.

Benefits of Hashing:

- Fast Access: Hashing allows for constant-time retrieval and insertion of data items, making it highly efficient for large datasets.
- Memory Efficiency: Hash tables use a fixed-size array, resulting in efficient memory utilization.
- Indexing: Hashing provides a way to index and search data based on a unique identifier (hash value), which can improve search performance.

Challenges of Hashing:

- Collisions: Hashing may result in collisions when two or more data items produce the same hash value. Collision resolution techniques are employed to handle these collisions effectively.
- Hash Function Design: Choosing an appropriate hash function that produces a uniform distribution of hash values is essential for minimizing collisions and maximizing efficiency.
- Hash Table Size: The size of the hash table affects the performance of hashing. If the table is too small, collisions may occur more frequently, while a large table may waste memory.

Overall, hashing is a powerful technique used in various applications, including database indexing, data retrieval, and caching. It provides efficient data organization and retrieval based on hash values, enabling quick access to data items.

Indexes

Indexes are data structures used in databases to improve the efficiency of data retrieval operations. They provide faster access to specific data records by creating a mapping between the values of one or more columns (search key) and

the physical location of the corresponding records. There are different types of indexes, including single-level indexes, multi-level indexes, B-tree indexes, and B+ tree indexes. Let's explore each of them in detail:

Single-Level Indexes:
Single-level indexes, also known as primary indexes or clustered indexes, directly map the search key values to the physical location of the records. They are based on a one-to-one mapping between the index entries and the data records.

Example of a single-level index on a primary key:

```
Table: Employees
-----------------------------------------
| EmployeeID | Name       | Department |
-----------------------------------------
| 1001       | John Smith | Sales      |
| 1002       | Lisa Brown | HR         |
| 1003       | Mike Jones | Finance    |
-----------------------------------------

Primary Index (EmployeeID):
--------------------------------------
| EmployeeID | Block Address         |
--------------------------------------
| 1001       | Address of block 1    |
| 1002       | Address of block 2    |
| 1003       | Address of block 3    |
--------------------------------------

In the above example, the primary index on the EmployeeID colu
mn directly maps the EmployeeID values to the physical block a
ddresses where the corresponding records are stored.
```

Multi-Level Indexes:
Multi-level indexes are designed to handle larger databases by organizing the index entries into multiple levels or layers. They provide efficient access to data by reducing the number of disk reads required to locate specific records.

Example of a multi-level index:

```
Table: Employees
-----------------------------------------
| EmployeeID | Name       | Department |
-----------------------------------------
| 1001       | John Smith | Sales      |
| 1002       | Lisa Brown | HR         |
| 1003       | Mike Jones | Finance    |
| ...        | ...        | ...        |
-----------------------------------------

Multi-Level Index (B+ Tree):
-------------------------------------------------
| Level 0          | Level 1             | Leaf Level |
-------------------------------------------------
| (1001, P1)      | (1001, P1)        | (1001, R1) |
| (1002, P2)      | (1002, P2)        | (1002, R2) |
| (1003, P3)      | (1003, P3)        | (1003, R3) |
| ...             | ...               | ...        |
-------------------------------------------------

In the above example, a multi-level index (specifically, a B+
tree) is used. The index has multiple levels, with the leaf le
vel containing the actual data records' references (R1, R2, R
3).
```

B-Tree Indexes:

B-tree indexes are balanced tree structures that store index entries in a specific order for efficient searching and retrieval. They are commonly used in databases and file systems. B-trees are self-balancing and can handle a large number of entries.

B+ Tree Indexes:
B+ tree indexes are a variant of B-tree indexes that further optimize range queries

and sequential access. In B+ trees, the leaf nodes form a linked list, allowing efficient scanning of the index entries.

Both B-tree and B+ tree indexes are multi-level indexes that provide efficient access to data by navigating through the index levels and narrowing down the search space.

Consider a table called "Students" with the following structure:

```
Table: Students
-----------------------------------
| StudentID | Name     | Age       |
-----------------------------------
| 1001      | John     | 20        |
| 1002      | Lisa     | 22        |
| 1003      | Mike     | 21        |
| 1004      | Sarah    | 19        |
| 1005      | David    | 23        |
| 1006      | Emily    | 20        |
-----------------------------------
```

1. B-Tree Index:
   Let's create a B-tree index on the "StudentID" column:

```
B-Tree Index (StudentID):
-----------------------------------------
| Node 1: [1002] | Node 2: [1004, 1005] |
-----------------------------------------
```

In the B-tree index, each node can contain multiple key values. The nodes are organized in a balanced manner to optimize search operations. The index indicates that Node 1 contains the key 1002, and Node 2 contains the keys 1004 and 1005. This index structure enables efficient search operations based on the StudentID values.

1. B+ Tree Index:
   Let's create a B+ tree index on the "Age" column:

```
B+ Tree Index (Age):
-----------------------------------------
| Leaf Node 1: [19, 1004] | Leaf Node 2: [20, 1001, 1006] |
-----------------------------------------
```

In the B+ tree index, the leaf nodes form a linked list, providing efficient range queries and sequential access. The leaf nodes contain both the key values (Age) and the corresponding pointers (StudentID). In this example, Leaf Node 1 contains the keys 19 and 1004, while Leaf Node 2 contains the keys 20, 1001, and 1006. This index structure allows for efficient retrieval of records based on Age values.

Both B-tree and B+ tree indexes provide fast data access by minimizing the number of disk reads required to locate specific records. They are suitable for large databases with a high volume of data.

These index types improve the performance of data retrieval operations, especially when searching for specific values or ranges of values. However, they come with some overhead in terms of additional storage space and maintenance cost during data modifications.

Choosing the appropriate index type

depends on various factors, such as the size of the database, the nature of the data, and the types of queries performed on the data.

Object-Oriented Database Management Systems (OODBMS):

Object-Oriented Database Management Systems (OODBMS) are database management systems that support the storage, management, and retrieval of objects, which are data entities that encapsulate both data and behavior. OODBMS combines the principles of object-oriented programming with database management, allowing the direct representation of complex data structures and their relationships.

Key concepts of OODBMS include:

1. Objects: Objects are instances of classes or types in an object-oriented programming language. They encapsulate both data attributes and methods or functions that operate on that data. In an OODBMS, objects can be stored directly in the database and accessed using object-oriented query languages.
2. Class Hierarchy: Objects in an OODBMS are organized into a class hierarchy, similar to inheritance in object-oriented programming. Classes represent a blueprint or template for creating objects, and subclasses inherit properties and behaviors from their parent classes.
3. Encapsulation: Encapsulation refers to the ability of objects to encapsulate their data and methods together, providing data hiding and abstraction. In OODBMS, encapsulation allows objects to maintain their internal state and provide controlled access to their data and behavior.
4. Inheritance: Inheritance enables the creation of subclasses that inherit properties and behaviors from parent classes. In OODBMS, inheritance can be used to model relationships between objects, allowing for code reuse and specialization.
5. Polymorphism: Polymorphism allows objects of different types to be treated as instances of a common superclass. In OODBMS, polymorphism supports dynamic binding and runtime method dispatch, enabling flexible and extensible data modeling.

Distributed Database Management Systems (DDBMS):

Distributed Database Management Systems (DDBMS) are database management systems that store data across multiple computers or nodes connected in a network. DDBMS enables the distribution of data to improve performance, scalability, and fault tolerance.

Key concepts of DDBMS include:

1. Data Distribution: In a DDBMS, data is distributed across multiple nodes or sites in a network. Different data distribution strategies can be used, such as horizontal partitioning (splitting data by rows), vertical partitioning (splitting data by columns), or a combination of both.

2. Transparency: Transparency in DDBMS refers to hiding the complexities of data distribution from users and applications. Different types of transparency include location transparency (users are unaware of the physical location of data), replication transparency (users are unaware of data replication), and fragmentation transparency (users are unaware of data fragmentation).
3. Distributed Query Processing: DDBMS provides mechanisms to process queries that involve data stored on multiple nodes. Query optimization techniques are employed to minimize network communication and ensure efficient execution of distributed queries.
4. Concurrency Control: Concurrency control is crucial in DDBMS to maintain data consistency when multiple transactions access and modify the distributed data concurrently. Techniques such as locking, timestamp ordering, and optimistic concurrency control are used to manage concurrent access.
5. Distributed Transactions: Distributed transactions involve multiple operations on distributed data that must be executed atomically and reliably. DDBMS provides transaction management mechanisms, such as two-phase commit protocol, to ensure the consistency and durability of distributed transactions.
6. Fault Tolerance: DDBMS incorporates mechanisms to handle failures and ensure data availability in the presence of network outages or node failures. Techniques like replication, backup, and recovery are used to maintain data integrity and recover from failures.

Both OODBMS and DDBMS offer advanced capabilities beyond traditional relational database management systems, catering to specific requirements of complex data structures, object-oriented programming paradigms, and distributed computing environments.

updated: https://yashnote.notion.site/DBMS-6b305e48c54944cf9ba3964f20bb7968?pvs=4