

## Unit I – Introduction

1. Describe the roles and responsibilities of database administrators in managing a database system, emphasizing their role in ensuring data security and integrity.

Ans:

Database administrators (DBAs) play a crucial role in managing database systems, and their responsibilities cover a wide range of tasks, with a primary focus on ensuring data security and integrity. Here are the key roles and responsibilities of database administrators:

1. **\*\*Database Design and Planning:\*\***

- Collaborate with stakeholders to understand data requirements.
- Design and plan the database structure to meet performance, storage, and accessibility needs.

2. **\*\*Installation and Configuration:\*\***

- Install and configure database management systems (DBMS) based on organizational needs.
- Optimize settings for performance and security.

3. **\*\*Data Modeling:\*\***

- Create and maintain data models that define the structure of the database.
- Ensure that data models adhere to normalization principles for efficiency.

4. **\*\*Security Management:\*\***

- Implement and manage security measures to protect sensitive data.
- Assign user roles and permissions to control access to data.
- Monitor and audit user activities to identify and mitigate security risks.

5. **\*\*Backup and Recovery:\*\***

- Develop and implement backup and recovery strategies to prevent data loss.
- Regularly perform backups and test restoration processes.

6. **\*\*Performance Monitoring and Tuning:\*\***

- Monitor database performance and identify areas for improvement.
- Optimize queries and configurations to enhance overall system performance.

7. **\*\*Capacity Planning:\*\***

- Forecast database growth and plan for increased storage and processing requirements.
- Ensure the system can handle future data loads without degradation in performance.

8. **\*\*Data Migration and Integration:\*\***

- Facilitate the smooth migration of data between systems or database versions.

- Integrate databases with other applications to ensure seamless data flow.

9. **Patch and Upgrade Management:**

- Keep abreast of software updates and security patches.
- Plan and execute upgrades to maintain a secure and up-to-date database environment.

10. **Disaster Recovery Planning:**

- Develop and implement disaster recovery plans to minimize downtime in the event of system failures.
- Conduct regular drills to test the effectiveness of recovery procedures.

11. **Documentation:**

- Maintain comprehensive documentation of database structures, configurations, and processes.
- Ensure that documentation is up-to-date and accessible to relevant stakeholders.

12. **Data Integrity and Quality:**

- Enforce data integrity constraints to maintain accurate and reliable information.
- Implement data quality checks and cleansing processes as needed.

13. **Compliance:**

- Ensure that the database system complies with relevant regulatory requirements (e.g., GDPR, HIPAA).
- Stay informed about changes in data protection laws and adjust the system accordingly.

14. **Troubleshooting and Support:**

- Respond to user issues and troubleshoot problems related to the database.
- Provide support and guidance to users and developers on database-related matters.

2. Discuss the concept of a "view of data" in database systems, elaborating on how it impacts user experience and data accessibility.

Ans:

In the context of database systems, a "view of data" refers to a virtual table that is based on the result of a SELECT query applied to one or more base tables. Unlike physical tables, views do not store data themselves but provide a way to represent data stored in the underlying tables in a specific way. Views can be used to simplify complex queries, provide a layer of security, and offer a customized perspective on the data for different users or applications.

Here are several aspects to consider when discussing the concept of a "view of data" and its impact on user experience and data accessibility:

1. **\*\*Simplification and Abstraction:\*\***

- Views allow users to abstract away the complexities of underlying table structures and relationships. Users can interact with the data in a simplified manner, focusing only on the relevant information presented by the view.
- Complex joins, calculations, or data transformations can be encapsulated within a view, making it easier for users to query and retrieve the desired information without having to understand the underlying database schema intricacies.

2. **\*\*Security and Data Restriction:\*\***

- Views can be used to implement security measures by restricting the columns or rows visible to different users. This is achieved by defining views that only expose specific subsets of the data, based on the user's role or permissions.
- By using views to control data access, sensitive information can be shielded from unauthorized users, ensuring that each user sees only the data they are allowed to view.

3. **\*\*Customization for User Needs:\*\***

- Views enable the customization of data representation for different users or applications. One view may include additional calculated columns or aggregate functions to meet the specific requirements of a particular user or reporting tool.
- This customization enhances the user experience by tailoring data presentations to match the user's needs, leading to more effective decision-making.

4. **\*\*Performance Optimization:\*\***

- Views can also contribute to performance optimization by pre-computing and storing the results of complex queries. This can reduce the overhead of executing resource-intensive queries repeatedly by allowing users to access the precomputed results stored in the view.

5. **\*\*Data Integrity and Consistency:\*\***

- Views can be used to enforce business rules and maintain data consistency by filtering or transforming data as it is accessed. This ensures that users always interact with a consistent and accurate representation of the data.

6. **\*\*Data Accessibility and Integration:\*\***

- Views provide a unified and standardized interface to the underlying data, promoting data accessibility and integration. Users and applications can interact with views without needing to understand the underlying database structure, facilitating ease of use and integration into various systems.

In summary, the concept of a "view of data" in database systems plays a crucial role in shaping user experience and data accessibility by providing a simplified, secure, and customized perspective on the underlying data. It allows for abstraction, security, customization, performance optimization, and improved data integrity, contributing to a more efficient and user-friendly database environment.

3. Compare and contrast centralized and distributed database architectures, highlighting their advantages, disadvantages with suitable scenarios.

Ans:

Centralized and distributed database architectures represent two fundamentally different approaches to organizing and managing data within a system. Let's compare and contrast these two architectures, highlighting their advantages, disadvantages, and suitable scenarios.

#### **\*\*Centralized Database Architecture:\*\***

##### **1. \*\*Definition:\*\***

- In a centralized database architecture, all data is stored and managed in a single location or server.
- Typically, there is a central point of control for data management and access.

##### **2. \*\*Advantages:\*\***

- **\*\*Simplicity:\*\*** Centralized databases are generally simpler to design and maintain.
- **\*\*Data Consistency:\*\*** Since there's a single data repository, maintaining consistency is relatively straightforward.
- **\*\*Security:\*\*** It's often easier to implement security measures in a centralized system.

##### **3. \*\*Disadvantages:\*\***

- **\*\*Single Point of Failure:\*\*** The entire system relies on a single server; if it fails, the entire database becomes inaccessible.
- **\*\*Scalability Issues:\*\*** As data and user load increase, scaling a centralized system can become challenging.
- **\*\*Limited Geographic Distribution:\*\*** Access may be slower for users located far from the central server.

##### **4. \*\*Suitable Scenarios:\*\***

- **\*\*Small to Medium-Sized Enterprises:\*\*** Centralized architectures are often suitable for smaller organizations with relatively simple data management needs.
- **\*\*Low Complexity Requirements:\*\*** When the data structure is uncomplicated, and there's no need for extensive scalability.

## **\*\*Distributed Database Architecture:\*\***

### **1. \*\*Definition:\*\***

- In a distributed database architecture, data is distributed across multiple servers or nodes, often geographically dispersed.
- There is no single point of control; instead, control and management are distributed.

### **2. \*\*Advantages:\*\***

- **\*\*Fault Tolerance:\*\*** The system continues to operate even if some nodes fail, reducing the risk of a complete system failure.
- **\*\*Scalability:\*\*** Distributed databases can more easily scale horizontally by adding new nodes to handle increased loads.
- **\*\*Improved Performance:\*\*** Users can access data from a node that is geographically closer, reducing latency.

### **3. \*\*Disadvantages:\*\***

- **\*\*Complexity:\*\*** Designing and maintaining a distributed system is more complex due to the need for synchronization and coordination.
- **\*\*Data Consistency Challenges:\*\*** Ensuring consistent data across all nodes can be challenging.
- **\*\*Security Concerns:\*\*** Distributed systems may introduce additional security challenges.

### **4. \*\*Suitable Scenarios:\*\***

- **\*\*Large Enterprises:\*\*** Distributed databases are often preferred for large organizations with diverse data management requirements.
- **\*\*High Scalability Requirements:\*\*** When there is a need for easy scalability to handle growing data and user loads.
- **\*\*Geographically Distributed Users:\*\*** If users are spread across different locations, a distributed architecture can improve data access times.

4. List four significant differences between a file-processing system and a Database Management System.

Ans:

Feature	File-Processing System	Database Management System (DBMS)
<b>Data Redundancy</b>	High data redundancy, as each program may maintain its own data files.	Controlled data redundancy through normalization, minimizing duplication.
<b>Data Independence</b>	Limited data independence; changes in data structure may require modifications in multiple programs.	High data independence; changes in data structure typically do not affect application programs.
<b>Data Integrity</b>	Relies on application programs to ensure data integrity; consistency is not guaranteed.	Enforces data integrity through constraints, transactions, and referential integrity.
<b>Data Security</b>	Limited security features; access control is usually implemented at the file level. <div>↓</div>	Advanced security features, including user authentication, role-based access control, and encryption.

<b>Concurrent Access</b>	Limited support for concurrent access; typically, file locks are used to prevent conflicts.	Provides mechanisms for concurrent access, such as transactions and isolation levels, ensuring data consistency.
<b>Data Retrieval</b>	Retrieval is often cumbersome, requiring custom code for each application to navigate and extract data.	Standardized query languages (e.g., SQL) make data retrieval more efficient and flexible.
<b>Scalability</b>	Limited scalability; scaling up may require significant modifications to existing programs and file structures.	Designed for scalability; can handle growing data volumes and user loads more effectively.
<b>Data Relationships</b>	Limited support for defining and managing relationships between different sets of data.	Allows the creation and management of complex relationships between tables using keys and foreign keys.
<b>Data Redundancy Control</b>	Minimal control over redundancy; each program manages its data, leading to duplication.	Reduces redundancy through normalization, minimizing data duplication and improving efficiency.
<b>Data Integrity Maintenance</b>	Relies on individual programs to maintain data integrity, increasing the risk of inconsistencies.	Enforces data integrity through the use of constraints, triggers, and stored procedures.
<b>Flexibility</b>	Limited flexibility; changes to data structures or business rules may require extensive modifications.	Offers greater flexibility as changes in data structure can be accommodated with minimal impact on applications.
<b>Backup and Recovery</b>	Backup and recovery procedures are typically manual and program-dependent.	Provides automated backup and recovery mechanisms, ensuring data consistency and system reliability.

5. What are the five main functions of a database administrator?

Ans: same as Q1

6. List three common challenges in database design and briefly describe how they can be addressed.

Ans:

1. **Normalization Challenges:**

- **Issue:** Ensuring that the database is properly normalized can be challenging. Over-normalization can lead to complex queries and decreased performance, while under-normalization can result in data redundancy and inconsistency.
- **Addressing the Challenge:** Strike a balance between normalization and denormalization based on the specific needs of the application. Consider factors such as query performance, ease of maintenance, and the nature of the data.

2. **Performance Optimization:**

- **Issue:** Achieving optimal performance for database queries can be challenging, especially as the volume of data grows. Inefficient queries, lack of indexes, or poor database design can contribute to slow performance.
- **Addressing the Challenge:** Regularly analyze and optimize queries by using appropriate indexes, caching mechanisms, and query optimization techniques. Monitor database performance and make adjustments to the design or configuration as needed.

3. **Data Security:**

- **Issue:** Protecting sensitive data and ensuring secure access is a critical challenge. Unauthorized access, data breaches, and other security vulnerabilities can pose significant risks to the integrity and confidentiality of the data.
- **Addressing the Challenge:** Implement robust authentication and authorization mechanisms. Encrypt sensitive data, both in transit and at rest. Regularly audit and monitor database activity to detect and respond to potential security threats. Adhere to best practices for database security and stay informed about emerging threats and security updates.

7. Describe five disadvantages of file systems.

Ans:

1. **Limited Scalability:**

- File systems can face scalability challenges as the amount of data grows. Managing a large number of files and directories can lead to decreased performance and increased complexity.

2. **Lack of Centralized Control:**

- File systems typically lack centralized control and may not provide robust mechanisms for access control and permissions. This can lead to security vulnerabilities and difficulties in managing user access across a network.

3. **Limited Metadata Handling:**



- File systems may have limitations in handling metadata, such as extended attributes and complex file relationships. This can restrict the ability to organize and search for files based on attributes other than just file names.

#### 4. **\*\*Fragmentation Issues:\*\***

- Fragmentation occurs when files are broken into pieces scattered across a storage device. This can lead to slower read and write speeds as the system must navigate and assemble fragmented data, impacting overall performance.

#### 5. **\*\*Difficulty in Version Control:\*\***

- File systems may not inherently support efficient version control mechanisms. Tracking changes to files and maintaining a history of modifications can be challenging, making it difficult to revert to previous states or track collaborative edits effectively.

8. List four real world applications that most likely employed a database system to store persistent data.

Ans:

#### 1. **\*\*Online Retail Systems:\*\***

- In online retail systems, databases are crucial for storing and managing product information, customer data, order details, and inventory levels. The relational structure of databases allows for efficient querying and updating of information, ensuring accurate and real-time data for both customers and retailers.

#### 2. **\*\*Hospital Information Systems:\*\***

- Healthcare organizations use databases to store patient records, medical histories, treatment plans, and billing information. A database system allows healthcare professionals to access patient data securely, track treatment progress, and maintain a comprehensive record of medical information for each patient.

#### 3. **\*\*Banking Systems:\*\***


- Banks and financial institutions rely heavily on database systems to manage customer accounts, transactions, loans, and other financial activities. The consistency and integrity of a database are critical for ensuring accurate and secure financial transactions, as well as for regulatory compliance.

#### 4. **\*\*Human Resources Management Systems (HRMS):\*\***

- HRMS applications use databases to store employee information, payroll data, attendance records, performance evaluations, and other HR-related data. This helps organizations efficiently manage their workforce, streamline HR processes, and ensure that employee records are accurate and up-to-date.

9.Explain the difference between two-tier and three-tier architectures.

Ans:

Aspect	Two-Tier Architecture	Three-Tier Architecture
<b>Number of Tiers</b>	Two	Three
<b>Components</b>	- Client	- Client (Presentation Tier)
	- Database Server (Data Tier)	- Application Server (Business Logic Tier)
		- Database Server (Data Tier)
<b>Responsibilities</b>	- Presentation	- Presentation (User Interface)
	- Data Access	- Business Logic (Application Processing)
		- Data Access (Communication with Database)
<b>Communication</b>	- Direct communication between client	- Client communicates with the Application
	and database server 	Server, which then communicates with the

		Database Server
<b>Scalability</b>	- Limited scalability as both presentation	- Improved scalability as each tier can be
	and data access are on the client side	scaled independently
<b>Flexibility</b>	- Less flexible as changes in one tier may	- More flexible as changes in one tier do
	affect the other	not necessarily impact the other tiers
<b>Maintenance</b>	- Easier maintenance due to fewer components	- Maintenance might be more complex due to
	and direct communication	the separation of components
<b>Example Applications</b>	- Small-scale applications	- Enterprise-level applications, web-based
	with limited complexity ↓	applications, complex systems

10. What are constraints? Discuss with respect to Domain Constraints, Referential Integrity, Assertions and Authorization.

Ans:

Constraints in the context of databases are rules or conditions that are defined to maintain the integrity, accuracy, and consistency of the data stored in a database. They ensure that the data conforms to certain standards and prevents the occurrence of undesirable situations. Let's discuss constraints in the context of domain constraints, referential integrity, assertions, and authorization.

#### 1. **\*\*Domain Constraints:\*\***

- **\*Definition:\*** Domain constraints define the allowable values for a particular attribute or column in a database table. They ensure that the data stored in a field adheres to a specific data type, format, or range.

- **\*Example:\*** If you have a "DateOfBirth" column, a domain constraint may specify that the date should be in the format YYYY-MM-DD and should not be in the future.

#### 2. **\*\*Referential Integrity:\*\***

- **\*Definition:\*** Referential integrity ensures that relationships between tables are maintained. It typically involves foreign key constraints, where a foreign key in one table refers to the primary

key in another table. This constraint ensures that values in the foreign key column match values in the primary key column.

- \*Example:\* If you have a "Orders" table with a foreign key referencing the "Customers" table, referential integrity ensures that an order cannot be created for a non-existent customer.

### 3. \*\*Assertions:\*\*

- \*Definition:\* Assertions are conditions or rules that are specified at the database level to enforce business rules or any other requirements that cannot be expressed by using domain constraints or referential integrity alone.

- \*Example:\* An assertion might specify that the total quantity of items in stock across all warehouses should not be negative.

### 4. \*\*Authorization Constraints:\*\*

- \*Definition:\* Authorization constraints, also known as access control, define the permissions and privileges granted to users or roles in a database. They determine who can access, modify, or delete data within the database.

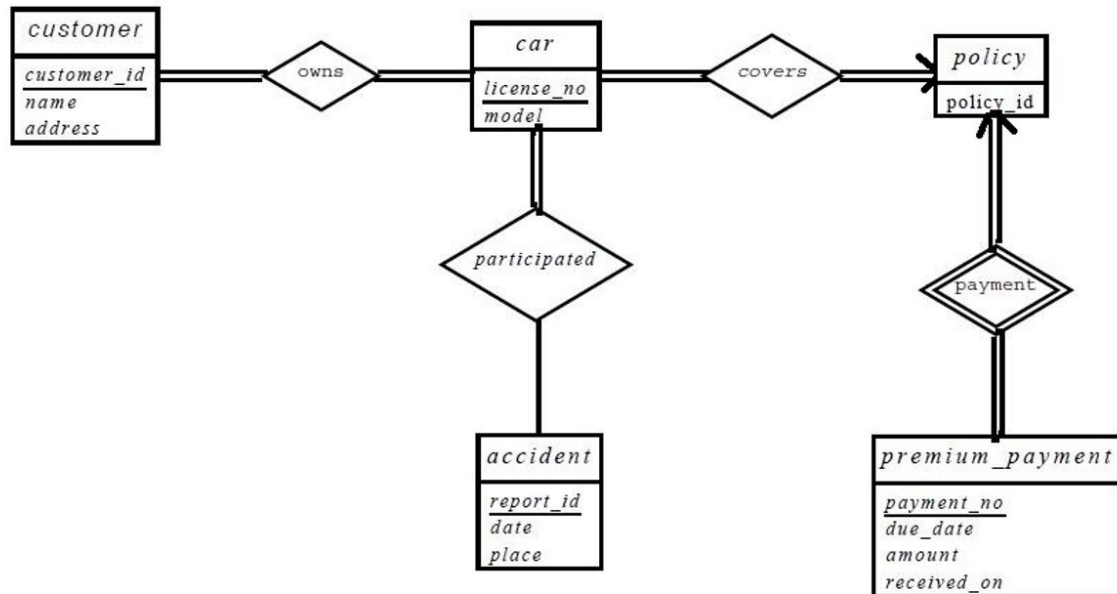
- \*Example:\* An authorization constraint might restrict a certain user from updating salary information in an employee database.

## **Unit II – Database Design and E-R Model**

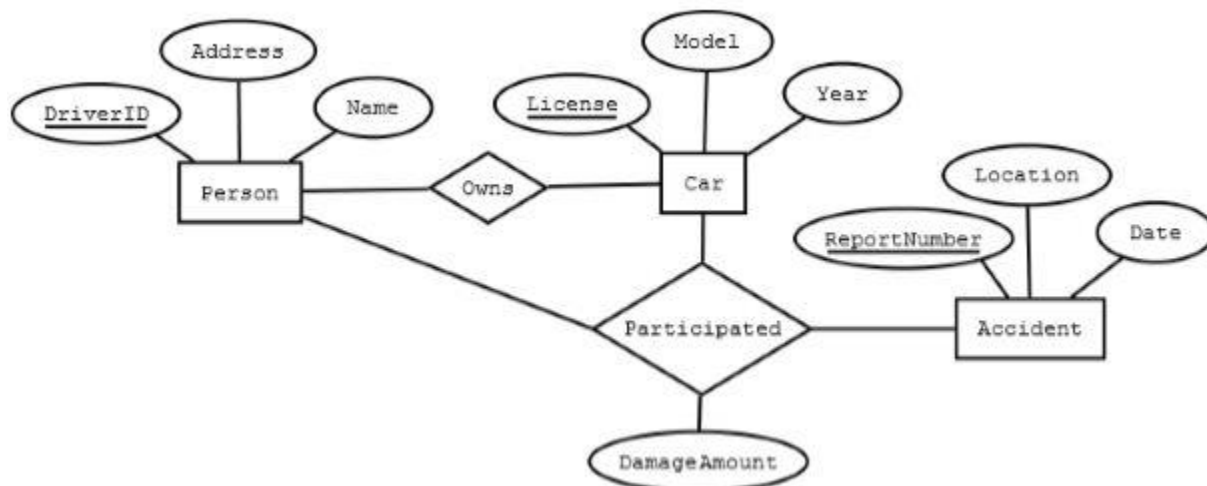
1. Construct an E-R diagram for a car insurance company whose customers own one or more cars each. Each car has associated with it zero to any number of recorded accidents. Each insurance policy covers one or more cars, and has one or more premium payments associated with it. Each payment is for a particular period of time, and has an associated due date, and the date when the payment was received.

Ans:

**Question 1.** Prepare a relational schema for the following requirements with associated E-R Diagram of a car insurance company.

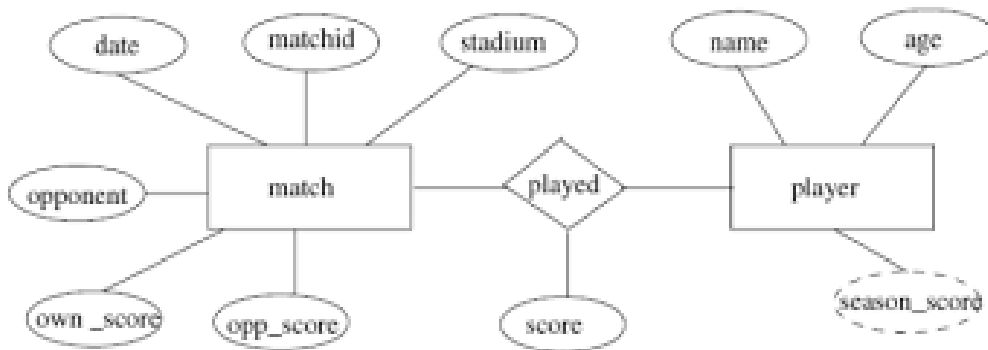


Customers own one or more cars. Each car has associated with it zero to any number of recorded accidents. Each insurance policy covers one or more cars and has one or more premium payments associated with it. Each payment is for a particular period of time and has an associated due date and the date the payment was received.

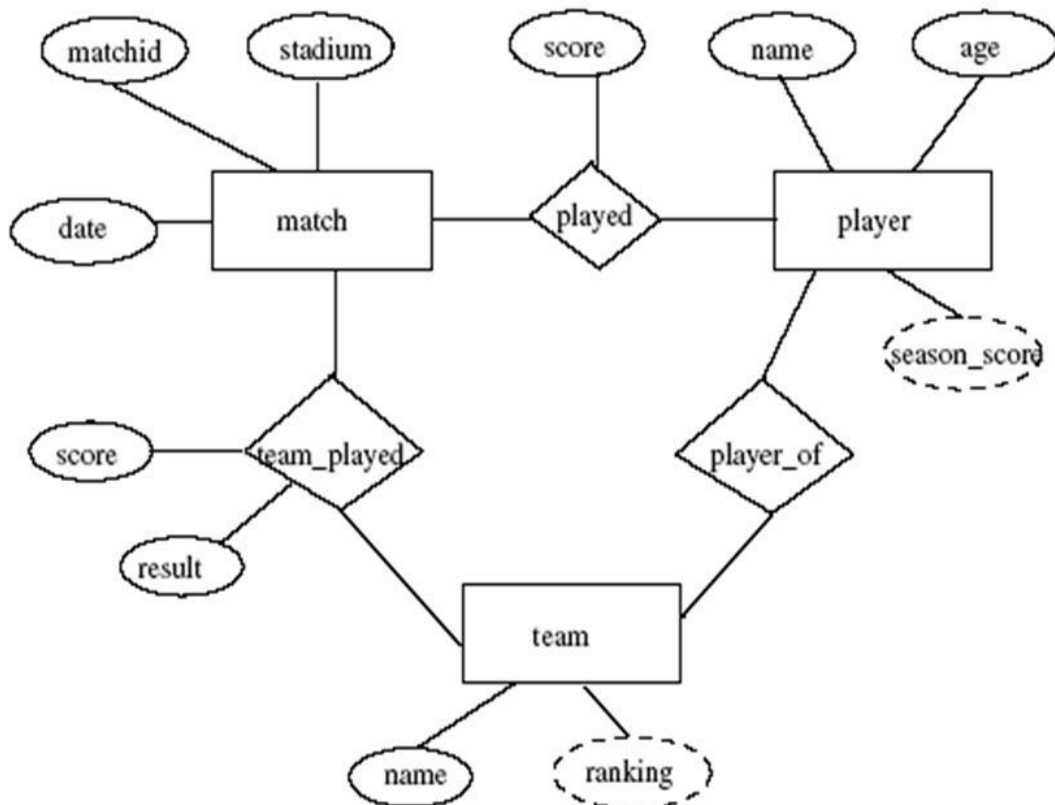


2.Design an E-R diagram for keeping track of the accomplishments of your favorite sports team. You should store the matches played, the scores in each match, the players in each match, and individual player statistics for each

match. Summary statistics should be modeled as derived attributes.

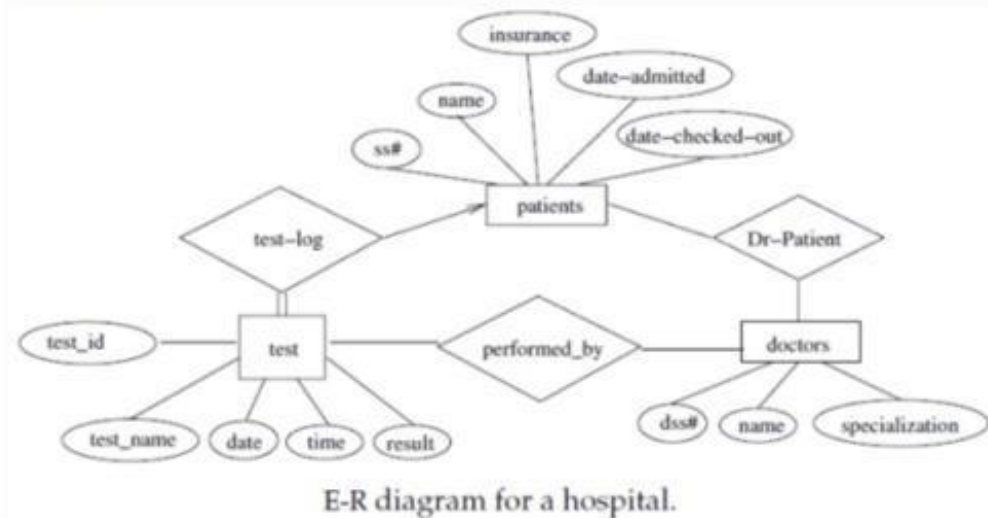


E-R diagram for favourite team statistics.



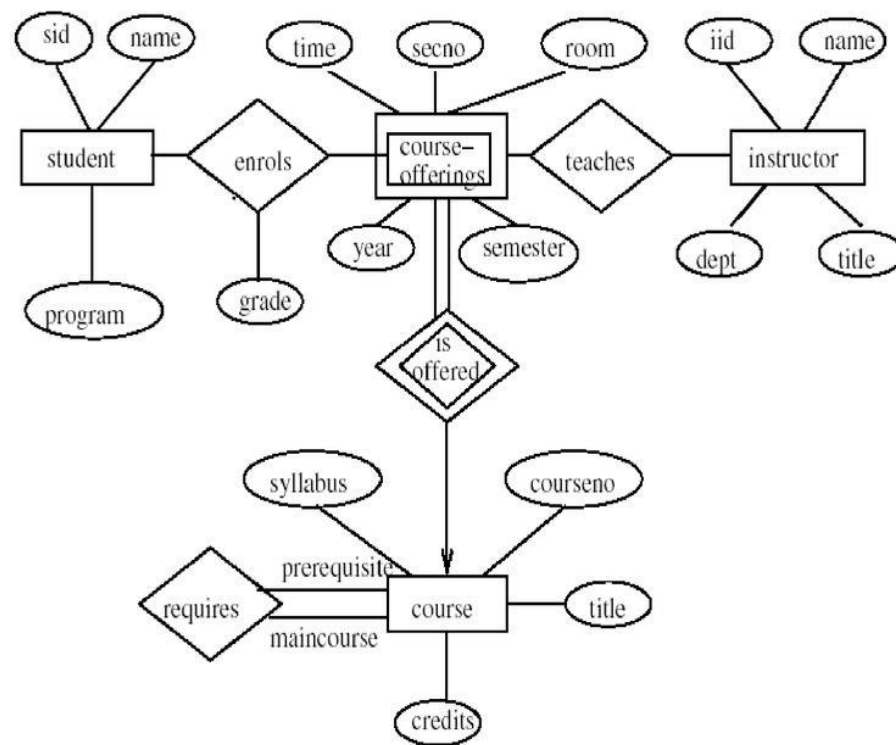
3. Construct an E-R diagram for a hospital with a set of patients and a set of medical doctors. Associate with each patient a log of the various tests and examinations conducted.

Answer :



4. A university registrar's office maintains data about the following entities: (a) courses including number, title, credits, syllabus, and prerequisites; (b) course offerings, including course number, year, semester, instructor(s), timings, and classroom; (c) students, including student-id, name, and program; and (d) instructors, including identification number, name, department, and title. Further, the enrollment of students in courses and grades awarded to students in each course they are enrolled for must be appropriately modeled. Construct an E-R diagram for the registrar's office. Document all assumptions that you make about the mapping constraints.

Ans:



Sunday, January 13,  
2019

5. Explain the concept of extended E-R features in the context of database design.

Ans:

In the context of database design, Extended Entity-Relationship (EER) features refer to enhancements made to the traditional Entity-Relationship (ER) model. The EER model introduces additional concepts and constructs to better represent complex relationships and constraints in a database system. These extensions are designed to provide a more expressive and powerful modeling capability compared to the basic ER model.

Here are some key extended E-R features:

#### 1. **Subtypes and Supertypes:**

- **Supertype:** A higher-level entity that represents a generalization of one or more subtypes.
- **Subtype:** Entities that inherit attributes from a supertype, representing specific categories or subgroups.

#### 2. **Specialization and Generalization:**



- **Specialization:** The process of defining subtypes based on certain characteristics or attributes.

- **Generalization:** The process of abstracting common features of entities to create a supertype.

### 3. **Category (Union) Types:**

- Allows an entity to belong to multiple entity types simultaneously. The attributes associated with a category type are a union of attributes from multiple entity types.

### 4. **Attribute Inheritance:**

- Subtypes inherit attributes from their supertype, facilitating the reuse of attributes and reducing redundancy.

### 5. **Aggregation:**

- Represents a relationship between a whole entity (aggregator) and its parts. It allows modeling complex relationships as a single unit.

### 6. **Multivalued Attributes:**

- Attributes that can have multiple values for a single entity. This helps in representing more complex data structures.

### 7. **Derived Attributes:**

- Attributes whose values can be derived from other attributes in the database. They are not stored explicitly but can be computed when needed.

### 8. **Associative Entities:**

- Intermediate entities introduced to represent relationships between two or more entities, each with its own set of attributes.

### 9. **Constraints:**

- EER models support additional constraints beyond those in the basic ER model, such as participation constraints, disjointness constraints, and completeness constraints.

### 10. **Roles in Relationships:**

- Allows entities to play different roles in different relationships, adding flexibility to the model.

Extended E-R features provide a more flexible and comprehensive way to model complex real-world scenarios, enabling database designers to capture a wider range of relationships and

constraints in their systems. These features enhance the precision and expressiveness of the data model, making it more reflective of the intricacies found in various domains.

6. Give an example of an E-R diagram and explain the components, including entities, attributes, and relationships.

Ans:

<https://www.tutorialspoint.com/what-are-the-components-of-er-diagrams-in-dbms>

7. Describe the process of reducing an E-R diagram to a relational schema.

Ans:

Reducing an Entity-Relationship (E-R) diagram to a relational schema involves transforming the conceptual model represented by the E-R diagram into a set of tables with defined relationships.

Here is a step-by-step process:

1. **\*\*Identify Entities:\*\***

- Each entity in the E-R diagram becomes a table in the relational schema. Identify all the entities and represent each one as a table with its attributes.

2. **\*\*Identify Attributes:\*\***

- For each entity, identify and list its attributes. Attributes become the columns of the corresponding table.

3. **\*\*Identify Primary Keys:\*\***

- Determine the primary key for each table. The primary key uniquely identifies each row in the table. In most cases, it corresponds to the entity's unique identifier.

4. **\*\*Handle Relationships:\*\***

- Identify relationships between entities in the E-R diagram. Relationships can be one-to-one, one-to-many, or many-to-many.

- Represent one-to-many relationships by including the primary key of the "one" side as a foreign key in the table of the "many" side.

- For many-to-many relationships, create a new table (association table) to represent the relationship, and include foreign keys from both related entities.

5. **\*\*Identify Foreign Keys:\*\***

- For each relationship, identify the foreign keys that reference the primary key of the related table. Foreign keys help establish referential integrity between tables.

6. **\*\*Resolve Weak Entities:\*\***

- If there are weak entities (entities that depend on another entity for their existence), resolve them by combining them with their parent entities. Attributes of the weak entity become part of the parent entity's table.

7. **\*\*Normalize Tables:\*\***

- Apply normalization techniques to ensure that the relational schema is in an appropriate normal form. Normalization helps in organizing data to minimize redundancy and dependency issues.

8. **\*\*Denormalization (if needed):\*\***

- In some cases, denormalization might be necessary for performance reasons. Evaluate whether to denormalize specific tables based on the system requirements.

9. **\*\*Document Constraints:\*\***

- Document any constraints, such as primary key constraints, foreign key constraints, and unique constraints, to ensure data integrity.

10. **\*\*Review and Refine:\*\***

- Review the relational schema to ensure that it accurately represents the information in the E-R diagram. Refine the schema based on any additional considerations or requirements.

The resulting relational schema provides a blueprint for creating a relational database that reflects the structure and relationships defined in the original E-R diagram.

8. Explain the concept of an E-R diagram. Describe the components and symbols used in an E-R diagram.

Ans:

An Entity-Relationship (E-R) diagram is a visual representation of the data model that represents the relationships between entities in a database. It is a popular tool in database design and is used to model the entities within a system and the relationships between them.

Components of an E-R Diagram:

1. **\*\*Entity:\*\***

- An entity is a real-world object or concept that has a distinct identity.
- It is represented by a rectangle in the diagram.
- Each entity is typically named and is used to store and organize information.

2. **\*\*Attribute:\*\***

- An attribute is a property or characteristic of an entity.
- It is represented by an oval and is connected to its respective entity.
- Attributes provide more detailed information about the entities.

3. **\*\*Relationship:\*\***

- A relationship illustrates how two or more entities are related to each other.
- It is represented by a diamond shape and is labeled to describe the nature of the relationship.
- Relationships can be one-to-one, one-to-many, or many-to-many.

#### 4. **Cardinality:**

- Cardinality defines the numerical relationships between entities in a relationship.
- It is often denoted by symbols like "1" or "M" to indicate the number of instances.
- For example, "1:1" signifies a one-to-one relationship, and "M:N" represents a many-to-many relationship.

#### 5. **Primary Key:**

- A primary key is a unique identifier for each record in an entity.
- It is usually underlined in the E-R diagram.

#### 6. **Foreign Key:**

- A foreign key is a field in one entity that refers to the primary key in another entity.
- It is used to establish a link between two entities in a relationship.

#### 7. **Weak Entity:**

- A weak entity is an entity that cannot be uniquely identified by its attributes alone; it depends on some related entity.
- It is represented by a double rectangle.

#### 8. **ISA (Inheritance):**

- ISA represents the inheritance relationship between entities.
- It is used when one entity is a specialized version of another.
- It is represented by a triangle connecting the specialized entity to the general entity.

#### Symbols Used:

- **Rectangles:** Represent entities.
- **Ovals:** Represent attributes.
- **Diamonds:** Represent relationships.
- **Lines:** Connect entities and show relationships.
- **Underlining:** Indicates the primary key.
- **Dashed Lines:** Indicate a weak relationship.

Overall, an E-R diagram provides a clear and concise way to visualize the structure of a database, making it easier to understand and communicate the relationships between different entities in a system.

9. Discuss the role of constraints in the E-R Model. How do constraints ensure data integrity and consistency in a database?

Ans:

In the Entity-Relationship (E-R) model, constraints play a crucial role in ensuring data integrity and consistency within a database. Constraints are rules or conditions applied to the data stored in a database to maintain accuracy, reliability, and coherence of the information. The E-R model uses various types of constraints to define relationships and dependencies between entities, attributes, and tables. Here are some key types of constraints in the E-R model and their roles in ensuring data integrity and consistency:

1. **Entity Integrity Constraints:**

- **Primary Key Constraint:** Every entity in the database must have a unique identifier known as the primary key. This ensures that each record within the entity can be uniquely identified, preventing duplicate entries and maintaining data integrity.

2. **Referential Integrity Constraints:**

- **Foreign Key Constraint:** This constraint establishes relationships between tables by linking the primary key of one table to the foreign key of another. It ensures that values in the foreign key column match values in the primary key column of the referenced table. This constraint helps maintain consistency by preventing orphaned records and enforcing relationships between related tables.

3. **Domain Constraints:**

- **Attribute Constraints:** Constraints applied to individual attributes to restrict the type, format, or range of values they can hold. This helps ensure that data entered into the database adheres to predefined rules, preventing data inconsistencies and errors.

4. **Business Rules Constraints:**

- **Check Constraints:** These constraints define specific conditions that must be met for data to be considered valid. They allow the database designer to enforce business rules and ensure that only appropriate data is stored, contributing to data integrity.

5. **Multiplicity Constraints:**

- **Cardinality Constraints:** Specify the number of instances of one entity that can be associated with another entity. This constraint ensures that relationships between entities are well-defined and helps maintain consistency in the data model.

Constraints in the E-R model collectively contribute to data integrity and consistency by:

- **Preventing Duplicate Data:** Entity integrity constraints, particularly primary keys, ensure that each record is unique.

- **Maintaining Relationships:** Referential integrity constraints guarantee that relationships between tables are well-maintained.
- **Enforcing Business Rules:** Check constraints and attribute constraints ensure that data adheres to predefined rules and standards.
- **Defining Relationship Cardinality:** Multiplicity constraints help in clearly defining and enforcing the relationships between entities.

By enforcing these constraints, the E-R model helps create a structured and reliable database that accurately represents the real-world relationships and ensures the quality of stored data.

10. Describe the Entity-Relationship (E-R) Model. What is the significance of entities, attributes, and relationships in this model? Provide an example of an E-R diagram to illustrate your explanation.

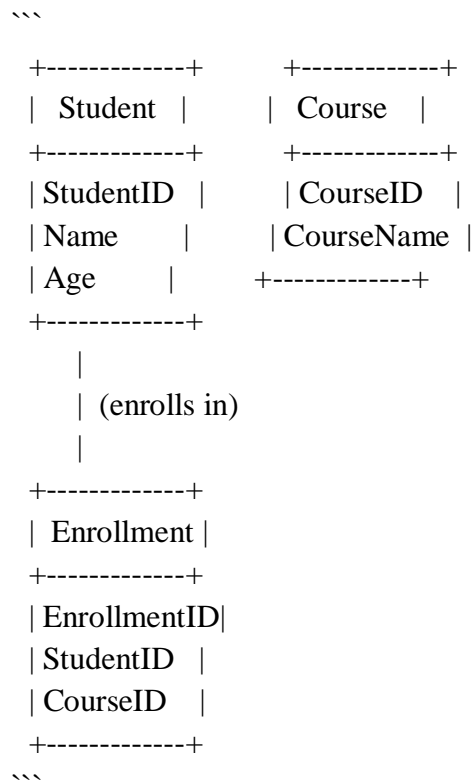
Ans:

The Entity-Relationship (E-R) Model is a conceptual data model used in database design to represent the relationships between different entities in a system. It provides a graphical representation of the data model, highlighting entities, attributes, and relationships. Here are the key components of the E-R Model:

1. **Entities:** Entities are objects or concepts in the real world that can be distinguished from one another. In a database, an entity is typically represented as a table. Each row in the table represents an instance of that entity, and each column represents an attribute of the entity.
2. **Attributes:** Attributes are properties or characteristics that describe an entity. For example, if "Person" is an entity, it might have attributes such as "Name," "Age," and "Address." Attributes help define the properties of an entity and are represented as columns in a table.
3. **Relationships:** Relationships describe how entities are related to each other. They are connections or associations between entities. Relationships can be one-to-one, one-to-many, or many-to-many. One-to-one means each entity in the relationship is related to only one other entity. One-to-many means one entity is related to multiple instances of another entity, and many-to-many means multiple instances of one entity are related to multiple instances of another entity.

Here's a simple example of an E-R diagram:

Consider two entities, "Student" and "Course," with attributes like "StudentID," "Name," "CourseID," and "CourseName." The relationship between them is that a student can enroll in multiple courses, and a course can have multiple students. The E-R diagram might look like this:



In this example:

- "Student" and "Course" are entities.
- "StudentID," "Name," "Age" are attributes of the "Student" entity.
- "CourseID" and "CourseName" are attributes of the "Course" entity.
- The "Enrollment" relationship indicates the association between students and courses, connecting the two entities. It has attributes like "EnrollmentID," "StudentID," and "CourseID."

This E-R diagram visually represents the structure of the database, showcasing entities, attributes, and the relationships between them. It serves as a blueprint for designing the actual relational database.

#### Short Answer Questions

1. What are constraints? Discuss with respect to Domain Constraints, Referential Integrity, Assertions and Authorization.
2. Describe types of attributes with examples.
3. Explain the difference between a weak and a strong entity set.
4. Define the concept of aggregation. Give two examples of where this concept is useful.
5. Discuss an E-R design issue related to handling multi-valued attributes in an entity.
6. Explain Union and Intersect operations in relational algebra with an example

for each.

7. Describe the use of aggregation in E-R modeling.
8. How does generalization/specialization play a role in an E-R model?
9. Explain ternary relationship in an E-R diagram with example.
10. Explain the distinction between total and partial participation.

1. **Constraints:**

- **Domain Constraints:** These specify the allowable values for attributes. For example, a domain constraint on an "age" attribute might specify that it must be a positive integer.
- **Referential Integrity:** Ensures that relationships between tables remain consistent. For instance, if a foreign key in one table refers to a primary key in another, referential integrity ensures that the foreign key always points to a valid primary key.
- **Assertions:** These are conditions that must be true for the database to be in a consistent state. For example, an assertion could enforce that the average salary of employees in a department should not exceed a certain limit.
- **Authorization:** Involves permissions and access controls to ensure that users can only perform actions for which they have permission.

2. **Types of Attributes:**

- **Simple Attribute:** Cannot be divided any further. Example: Age.
- **Composite Attribute:** Composed of multiple simple attributes. Example: Address (composed of street, city, and zip code).
- **Derived Attribute:** Derived from other attributes. Example: Age can be derived from the birthdate.
- **Multi-valued Attribute:** Can hold multiple values. Example: Phone numbers for a person.

3. **Weak vs. Strong Entity Set:**

- **Strong Entity Set:** Has a primary key and is not dependent on any other entity for its existence. It can exist independently.
- **Weak Entity Set:** Does not have a primary key and is existence-dependent on another entity, called the owner entity. It cannot exist without the owner entity.

4. **Aggregation:**

- Aggregation is a concept where relationships are treated as entities. It represents a 'whole-part' relationship.
- **Examples:**
  - In a university database, a "Department" entity may aggregate "Professor" entities.
  - In a car rental system, a "Rental" entity may aggregate "Car" entities.



5. **Handling Multi-valued Attributes:**

- Multi-valued attributes can be problematic in a relational database. One solution is to create a separate table for the multi-valued attribute, linked to the original entity through a foreign key.

6. **Union and Intersect Operations in Relational Algebra:**

- **Union Operation (  $\cup$  ):** Combines tuples from two relations and removes duplicates.

Example:  $(R \cup S)$

- **Intersect Operation (  $\cap$  ):** Retrieves common tuples from two relations. Example:  $(R \cap S)$

7. **Aggregation in E-R Modeling:**

- Aggregation in E-R modeling is used to represent relationships involving the 'whole-part' concept, similar to aggregation in database design.

8. **Generalization/Specialization in E-R Modeling:**

- **Generalization:** Combining multiple entity sets into a higher-level entity. Example: "Employee" generalizing "Manager" and "Clerk."

- **Specialization:** Creating specialized entities from a higher-level entity. Example: "Manager" and "Clerk" are specializations of the "Employee" entity.

9. **Ternary Relationship in E-R Diagram:**

- Involves three entities and describes a relationship between them. Example: A "Project" involves multiple "Employees" with specific "Tasks" assigned.

10. **Total vs. Partial Participation:**

- **Total Participation:** Every entity in the entity set must participate in the relationship. Represented by a double line connecting the entity to the relationship line.

- **Partial Participation:** Some entities in the entity set may not participate in the relationship. Represented by a single line connecting the entity to the relationship line.

## **Unit III – SQL and Advanced SQL**

Write SQL statements for the following.

A. Query all attributes of every Japanese city in the CITY table.

The COUNTRYCODE for Japan is JPN. (1M)

The CITY and STATION tables are described as follows:

STATION		CITY	
Field	Type	Field	Type
ID	NUMBER	ID	NUMBER
CITY	VARCHAR2(21)	NAME	VARCHAR2(17)
STATE	VARCHAR2(2)	COUNTRYCODE	VARCHAR2(3)
LAT_N	NUMBER	DISTRICT	VARCHAR2(20)
LONG_W	NUMBER	POPULATION	NUMBER

where LAT\_N is the northern latitude and LONG\_W is the western longitude.

B. Query all columns for all American cities in the CITY table with populations larger than 100000. The Country Code for America is USA. (2M)

C. Query a count of the number of cities in CITY having a Population larger than 100,000. (2M)

D. Query the sum of Northern Latitudes (LAT\_N) from STATION having values greater than 38.7880 and less than 137.2345. (2M)

Ans:

Certainly! Here are the SQL statements for the given queries:

A. Query all attributes of every Japanese city in the CITY table:

```
```sql
SELECT *
FROM CITY
WHERE COUNTRYCODE = 'JPN';
```
```

B. Query all columns for all American cities in the CITY table with populations larger than 100,000:

```
```sql
SELECT *
FROM CITY
WHERE COUNTRYCODE = 'USA' AND POPULATION > 100000;
```
```

C. Query a count of the number of cities in CITY having a Population larger than 100,000:

```
```sql
SELECT COUNT(*)
FROM CITY
WHERE POPULATION > 100000;
```
```

D. Query the sum of Northern Latitudes (LAT\_N) from STATION having values greater than 38.7880 and less than 137.2345:

```
```sql
SELECT SUM(LAT_N)
FROM STATION
WHERE LAT_N > 38.7880 AND LAT_N < 137.2345;
```
```

Write SQL statement for the following.

| CITY        |              | COUNTRY        |              |
|-------------|--------------|----------------|--------------|
| Field       | Type         | Field          | Type         |
| ID          | NUMBER       | CODE           | VARCHAR2(3)  |
| NAME        | VARCHAR2(17) | NAME           | VARCHAR2(44) |
| COUNTRYCODE | VARCHAR2(3)  | CONTINENT      | VARCHAR2(13) |
| DISTRICT    | VARCHAR2(20) | REGION         | VARCHAR2(25) |
| POPULATION  | NUMBER       | SURFACEAREA    | NUMBER       |
|             |              | INDEPYEAR      | VARCHAR2(5)  |
|             |              | POPULATION     | NUMBER       |
|             |              | LIFEEXPECTANCY | VARCHAR2(4)  |
|             |              | GNP            | NUMBER       |
|             |              | GNPOLD         | VARCHAR2(9)  |
|             |              | LOCALNAME      | VARCHAR2(44) |
|             |              | GOVERNMENTFORM | VARCHAR2(44) |
|             |              | HEADOFSTATE    | VARCHAR2(32) |
|             |              | CAPITAL        | VARCHAR2(4)  |
|             |              | CODE2          | VARCHAR2(2)  |

A. Query the list of CITY names ending with vowels (a, e, i, o, u) from STATION. Your result cannot contain duplicates. (2M)

B. Given the CITY and COUNTRY tables, query the sum of the populations of all cities where the CONTINENT is 'Asia'. (2M)  
The CITY and COUNTRY tables are described as follows:

Note: CITY.CountryCode and COUNTRY.Code are matching key columns.

C. Given the CITY and COUNTRY tables, query the names of all the continents (COUNTRY.Continent) and their respective average city populations (CITY.Population). (3M)

Ans:

Certainly! Here are the SQL statements for the given queries:

A. Query the list of CITY names ending with vowels (a, e, i, o, u) from STATION. Your result cannot contain duplicates:

```
```sql
SELECT DISTINCT NAME
FROM STATION
WHERE NAME LIKE '%a' OR NAME LIKE '%e' OR NAME LIKE '%i' OR NAME LIKE '%o'
OR NAME LIKE '%u';
```
```

B. Given the CITY and COUNTRY tables, query the sum of the populations of all cities where the CONTINENT is 'Asia':

```
```sql
SELECT SUM(CITY.Population)
FROM CITY
JOIN COUNTRY ON CITY.CountryCode = COUNTRY.Code
WHERE COUNTRY.Continent = 'Asia';
```
```

C. Given the CITY and COUNTRY tables, query the names of all the continents (COUNTRY.Continent) and their respective average city populations (CITY.Population):

```
```sql
SELECT COUNTRY.Continent, AVG(CITY.Population) AS AvgCityPopulation
FROM CITY
JOIN COUNTRY ON CITY.CountryCode = COUNTRY.Code
GROUP BY COUNTRY.Continent;
Write SQL statement for the following:
```

The Product table is described as follows:

PRODID	PDESC	PRICE	CATEGORY	DISCOUNT
101	Basketball	10	Sports	5
102	Shirt	20	Apparel	10
103	NULL	30	Electronics	15
104	Cricket Bat	20	Sports	20
105	Trouser	10	Apparel	5
106	Television	40	ELECTRONICS	20

- A. Write a query to display Product Id, Product Description and Category of those products whose category name is electronics. (1M)
- B. Write a query to display the product id, first five characters of product description and category of products. (1M)
- C. Write a query to display the product description and discount for all the products. Display the value 'No Description' if description is not having any value i.e. NULL. (1M)
- D. Display product id, category, price and discount of all products in descending order of category and ascending order of price. (2M)
- E. Display product id, category and discount of the products which belongs to the category 'Sports' or 'Apparel' in ascending order of category and discount. (2M)

Ans:

Certainly! Here are the SQL statements for the given queries:

- A. Write a query to display Product Id, Product Description, and Category of those products whose category name is electronics:

```
```sql
SELECT ProductId, ProductDescription, Category
FROM Product
WHERE Category = 'Electronics';
```
```

B. Write a query to display the product id, first five characters of product description, and category of products:

```
```sql
SELECT ProductId, LEFT(ProductDescription, 5) AS ShortDescription, Category
FROM Product;
```
```

C. Write a query to display the product description and discount for all the products. Display the value 'No Description' if the description is not having any value i.e. NULL:

```
```sql
SELECT COALESCE(ProductDescription, 'No Description') AS Description, Discount
FROM Product;
```
```

D. Display product id, category, price, and discount of all products in descending order of category and ascending order of price:

```
```sql
SELECT ProductId, Category, Price, Discount
FROM Product
ORDER BY Category DESC, Price ASC;
```
```

E. Display product id, category, and discount of the products that belong to the category 'Sports' or 'Apparel' in ascending order of category and discount:

```
```sql
SELECT ProductId, Category, Discount
FROM Product
WHERE Category IN ('Sports', 'Apparel')
ORDER BY Category ASC, Discount ASC;
```

Refer assignments for onwards questions

1. Describe the significance of primary keys in a relational database with example.
2. Explain the concept of referential integrity in the context of database constraints.

3. What is the difference between DROP and DELETE statements? Explain with examples.
4. What are aggregate functions? Give examples.
5. In SQL, when you execute a query, the SQL clauses are processed in a specific order. Describe the typical order of execution for SQL clauses and explain why this order is important in query processing.
6. Compare SQL set operations UNION, INTERSECT, and EXCEPT. Give example for each.
7. What are integrity constraints in SQL, and why are they essential in database design and management? Explain any one constraint with example.
8. Differentiate between UNION and UNION ALL set operations in SQL.
9. Compare and contrast the use of INNER JOIN and LEFT JOIN in SQL with examples.
10. Discuss the differences between INNER JOIN and OUTER JOIN in SQL. Give example for each

Ans:

1. **\*\*Significance of Primary Keys:\*\***

- A primary key is a unique identifier for each record in a relational database table.
- It ensures the uniqueness of each row, helping to distinguish one record from another.
- Primary keys are crucial for maintaining data integrity and facilitating relationships between tables.
- Example: In a "Students" table, the "student\_id" column can be set as the primary key. Each student will have a unique student ID, and this key is used to establish relationships with other tables, such as "Courses" where it might be a foreign key.

2. **\*\*Referential Integrity:\*\***

- Referential integrity ensures that relationships between tables remain consistent.
- It is maintained through foreign key constraints, ensuring that values in one table's foreign key match the values in another table's primary key.
- If a foreign key references a primary key, the values in the foreign key must exist in the referenced primary key.
- Example: If a "Orders" table has a foreign key "customer\_id" referencing the "Customers" table's primary key, it ensures that every order is associated with an existing customer.

3. **\*\*DROP vs. DELETE Statements:\*\***

- `DROP` is used to delete an entire table or a database.
- Example: `DROP TABLE Employees;`
- `DELETE` is used to remove specific rows from a table based on a condition.
- Example: `DELETE FROM Employees WHERE salary < 50000;`

#### 4. **\*\*Aggregate Functions:\*\***

- Aggregate functions perform a calculation on a set of values and return a single value.
- Examples: `SUM()`, `AVG()`, `COUNT()`, `MIN()`, `MAX()`.
- Example: `SELECT AVG(salary) FROM Employees;`

#### 5. **\*\*Order of Execution for SQL Clauses:\*\***

- 1. `FROM` - Specifies the tables involved.
- 2. `WHERE` - Filters rows based on conditions.
- 3. `GROUP BY` - Groups rows that share common values.
- 4. `HAVING` - Filters groups based on conditions.
- 5. `SELECT` - Retrieves the specified columns.
- 6. `ORDER BY` - Sorts the result set.
- The order is important for logical query processing and efficiency.

#### 6. **\*\*Set Operations in SQL:\*\***

- `UNION` - Combines results of two queries, removing duplicates.
- Example: `SELECT column1 FROM table1 UNION SELECT column1 FROM table2;`
- `INTERSECT` - Returns common rows between two queries.
- Example: `SELECT column1 FROM table1 INTERSECT SELECT column1 FROM table2;`
- `EXCEPT` - Returns rows from the first query that are not in the second.
- Example: `SELECT column1 FROM table1 EXCEPT SELECT column1 FROM table2;`

#### 7. **\*\*Integrity Constraints in SQL:\*\***

- Integrity constraints enforce data accuracy and consistency.
- Example: `CHECK` constraint ensures that a value in a column meets a specific condition.
- `ALTER TABLE Employees ADD CONSTRAINT check\_salary CHECK (salary > 0);`

#### 8. **\*\*UNION vs. UNION ALL:\*\***

- `UNION` removes duplicate rows; `UNION ALL` retains all rows.
- Example: `SELECT column1 FROM table1 UNION SELECT column1 FROM table2;`
- Example: `SELECT column1 FROM table1 UNION ALL SELECT column1 FROM table2;`

#### 9. **\*\*INNER JOIN vs. LEFT JOIN:\*\***

- `INNER JOIN` returns matching rows from both tables.
- Example: `SELECT \* FROM Employees INNER JOIN Departments ON Employees.department\_id = Departments.department\_id;`
- `LEFT JOIN` returns all rows from the left table and matching rows from the right.
- Example: `SELECT \* FROM Employees LEFT JOIN Departments ON Employees.department\_id = Departments.department\_id;`



10. **\*\*INNER JOIN vs. OUTER JOIN:\*\***

- `INNER JOIN` returns only matching rows.

- Example: `SELECT \* FROM Employees INNER JOIN Departments ON Employees.department\_id = Departments.department\_id;`

- `OUTER JOIN` is a general term; `LEFT JOIN` and `RIGHT JOIN` are specific types.

- `LEFT JOIN` returns all from the left table and matching from the right.

- Example: `SELECT \* FROM Employees LEFT JOIN Departments ON Employees.department\_id = Departments.department\_id;`

## **Unit IV – Relational Database Design**

1. What is lossy and lossless decomposition? Consider the following table

ID name street city salary

12345 ABC Main Perryridge 75000

67890 ABC North Hampton 67000

Decompose this relation as lossy as well as lossless decomposition

Ans:

In the context of database normalization, decomposition refers to the process of breaking down a relation (table) into multiple smaller relations to eliminate redundancy and dependency issues. Decomposition can be classified as lossy or lossless, depending on whether information is lost during the process.

1. **\*\*Lossless Decomposition:\*\***

Lossless decomposition ensures that no information is lost during the process. This is achieved by creating new relations (tables) in a way that, when joined, they can reconstruct the original relation. Common techniques for lossless decomposition include the Boyce-Codd Normal Form (BCNF) and the Third Normal Form (3NF).

Given your table:

---

ID name street city salary

12345 ABC Main Perryridge 75000

67890 ABC North Hampton 67000

---

A lossless decomposition could involve splitting the table into two tables based on functional dependencies. For example:

- Table 1: Employees

---

ID	name
12345	ABC
67890	ABC
...	

- Table 2: EmployeeDetails

ID	street	city	salary
12345	Main	Perryridge	75000
67890	North Hampton		67000
...			

In this case, the original table can be reconstructed by joining the "Employees" and "EmployeeDetails" tables using the common attribute "ID."

## 2. \*\*Lossy Decomposition:\*\*

Lossy decomposition involves breaking down a relation in a way that may result in the loss of information. This is generally not desirable but may be necessary in certain situations. An example of a lossy decomposition might involve grouping attributes in a way that causes information to be lost when the original table cannot be precisely reconstructed.

In this specific case, a lossy decomposition might involve combining some attributes into a single table, thereby losing the original structure. For instance:

- Lossy Table:

ID	name_street_city	salary
12345	ABC_Main Perryridge	75000
67890	ABC_North Hampton	67000
...		

In this lossy decomposition, the original attributes "name," "street," and "city" are combined into a single attribute "name\_street\_city." While this reduces redundancy, it may not be possible to precisely reconstruct the original table from this decomposed form.

It's important to note that the choice between lossless and lossy decomposition depends on the specific requirements of the application and the trade-offs between minimizing redundancy and preserving information.

2. What is 1NF? Consider the following tables and check whether the following table is in 1NF. If not then normalize the database to 1NF.

DPT\_NO MG\_NO EMP\_NO EMP\_NM

D101 12345 20000

20001

20002

Carl Sagan

Mag James

Larry Bird

D102 13456 30000

30001

Jim Carter

Paul Simon

Ans:

1NF (First Normal Form) is a property of a relation in a relational database. A table is said to be in 1NF if it contains only atomic (indivisible) values, and there are no repeating groups or arrays. Each column of the table must contain only a single value, and each value must be atomic.

Let's analyze the given table:

---

DPT\_NO | MG\_NO | EMP\_NO | EMP\_NM

-----

D101 | 12345 | 20000 | Carl Sagan

| | 20001 | Mag James

| | 20002 | Larry Bird

D102 | 13456 | 30000 | Jim Carter

| | 30001 | Paul Simon

---

It appears that the table is not in 1NF because the "EMP\_NO" and "EMP\_NM" columns have repeating groups, and they represent a multivalued attribute. To normalize this table to 1NF, you can create a separate table for employees:

Table 1: Department (DPT\_NO, MG\_NO)

---

DPT\_NO | MG\_NO

-----

D101 | 12345

D102 | 13456

...

Table 2: Employee (EMP\_NO, EMP\_NM)

...

EMP\_NO | EMP\_NM

-----

20000 | Carl Sagan

20001 | Mag James

20002 | Larry Bird

30000 | Jim Carter

30001 | Paul Simon

...

Now, both tables are in 1NF. Each column contains atomic values, and there are no repeating groups. The "EMP\_NO" and "EMP\_NM" attributes have been separated into a different table, removing the multivalued attribute and adhering to the first normal form.

3. Define and illustrate the term functional dependency with example. Given R {ABCD} and a set F of functional dependencies on R given as

$F = \{AB \rightarrow C,$

$AB \rightarrow D,$

$C \rightarrow A,$

$D \rightarrow B \}$ .

Find any two candidate keys of R.

Ans:

In the context of relational databases, a functional dependency is a constraint between two sets of attributes in a relation. Specifically, if the values of one set of attributes uniquely determine the values of another set, we say there is a functional dependency between them.

Let's consider the given relation  $(R = \{A, B, C, D\})$  and the set of functional dependencies  $(F = \{AB \rightarrow C, AB \rightarrow D, C \rightarrow A, D \rightarrow B\})$ .

Here,  $(AB \rightarrow C)$  means that knowing the values of attributes A and B uniquely determines the value of attribute C, and similarly for the other functional dependencies.

To find candidate keys, we need to identify a set of attributes that uniquely determines all other attributes in the relation. One way to approach this is to find the closure of different attribute sets with respect to the given functional dependencies.

Let's start by considering individual attributes and checking if they are superkeys:

1. **\*\*Check {A} for superkey:\*\***
  - $\{A\}^+ = \{A\}$  (by reflexive rule)
  - $\{A\}$  is not a superkey as it does not cover all attributes.
2. **\*\*Check {B} for superkey:\*\***
  - $\{B\}^+ = \{B\}$  (by reflexive rule)
  - $\{B\}$  is not a superkey as it does not cover all attributes.
3. **\*\*Check {C} for superkey:\*\***
  - $\{C\}^+ = \{C, A\}$  (using  $C \rightarrow A$ )
  - $\{C\}$  is not a superkey as it does not cover all attributes.
4. **\*\*Check {D} for superkey:\*\***
  - $\{D\}^+ = \{D, B\}$  (using  $D \rightarrow B$ )
  - $\{D\}$  is not a superkey as it does not cover all attributes.
5. **\*\*Check {A, B} for superkey:\*\***
  - $\{A, B\}^+ = \{A, B, C, D\}$  (using  $AB \rightarrow C, AB \rightarrow D$ )
  - $\{A, B\}$  is a superkey as it covers all attributes.

Thus,  $\{A, B\}$  is one candidate key.

Now, let's check if there's another candidate key:

6. **\*\*Check {C, D} for superkey:\*\***
  - $\{C, D\}^+ = \{C, D, A, B\}$  (using  $C \rightarrow A, D \rightarrow B$ )
  - $\{C, D\}$  is a superkey as it covers all attributes.

So, another candidate key is  $\{C, D\}$ .

Therefore, two candidate keys for relation R are  $\{A, B\}$  and  $\{C, D\}$ .

What is lossy and lossless decomposition? Consider the following table

ID	name	street	city	salary
57766	Kim	Main	Perryridge	75000
98776	Kim	North	Hampton	67000

Decompose this relation as lossy as well as lossless decomposition

Ans:

Lossy and lossless decomposition are concepts in database normalization, particularly in the context of relational databases.

### 1. **\*\*Lossless Decomposition:\*\***

- In lossless decomposition, we aim to decompose a relation into multiple smaller relations without losing any information. This means that we should be able to reconstruct the original relation by joining the decomposed relations.
- One common approach to achieve lossless decomposition is to use functional dependencies and the process of normalization. Functional dependencies describe the relationships between attributes in a relation.
- Given your table:

```

...
ID   | name | street   | city     | salary
-----|-----|-----|-----|-----
57766 | Kim  | Main Street | Perryridge | 75000
98776 | Kim  | North Street | Hampton   | 67000
...
```

- Assuming that  $\{ID\} \rightarrow \{name, street, city, salary\}$  is a superkey (a key), there is no lossless decomposition needed because the table is already in a lossless form.

### 2. **\*\*Lossy Decomposition:\*\***

- In lossy decomposition, we intentionally discard some information while decomposing a relation. This may be done to achieve certain desirable properties such as reducing redundancy or improving performance.
- Lossy decomposition can result in some loss of information, and it may not always be possible to reconstruct the original relation completely.
- An example of a lossy decomposition of your table might involve splitting it into two relations, one with  $\{ID, name, street\}$  and another with  $\{ID, city, salary\}$ . However, if you only join these two relations, you won't be able to recreate the original table because there is no common attribute between them.

Example:

...

Relation 1: ID | name | street

57766 | Kim | Main Street

98776 | Kim | North Street

Relation 2: ID | city | salary

57766 | Perryridge | 75000

98776 | Hampton | 67000

...

In this lossy decomposition, if you join the two relations on the common attribute `ID`, you can't get back the original table since the attribute `name` is missing from Relation 2.

In summary, for your provided table, a lossless decomposition is already achieved as it is in a well-normalized form. Lossy decomposition would involve intentionally breaking the table in a way that prevents complete reconstruction.

Student ID	Name	Contact No	Course	Marks	Grade
1	James Potter	111-111-1111, 123-456-7890	OOP	80	B+
1	James Potter	111-111-1111, 123-456-7890	DBMS	95	A+
2	Ethan McCarty	222-222-3222	OOP	75	B
2	Ethan McCarty	222-222-3222	SE	85	A
3	Emily Rayner	333-333-3333	PF	75	B

Client No	Client Name	Property No	Address	Rent Duration	Rent	Owner No	Owner Name
C1	James Mathew	P1	Electronic City, Mysore	01-Jan-13 to 09-Sep-13	1000	O101	Tina Joseph
C1	James Mathew	P2	Sankranthi Circle, Mysore	01-Sep-13 to 01-Dec-14	800	O109	Dinesh Shah
C2	Arvind Gupta	P8	Kormangla, Bangalore	01-Dec-13 to 01-Jan-15	1500	O105	Ram

First Normal Form (1NF) is a property of a relation in a relational database. A table is said to be in 1NF if it contains only atomic (indivisible) values, and there are no repeating groups or arrays of values.

Let's check the given tables for 1NF:

### Table 1: Student

...

Student ID	Name	Contact No	Course	Marks	Grade
1	James Potter	111-111-1111, 123-456-7890	OOP	80	B+
1	James Potter	111-111-1111, 123-456-7890	DBMS	95	A+
2	Ethan McCarty	222-222-3222	OOP	75	B
2	Ethan McCarty	222-222-3222	SE	85	A
3	Emily Rayner	333-333-3333	PF	75	B

...



The "Contact No" column violates 1NF because it contains multiple phone numbers. To bring this table to 1NF, we need to create a new table for contact information.

### Updated Table 1: Student

---

Student ID	Name	Contact No
1	James Potter	111-111-1111
1	James Potter	123-456-7890
2	Ethan McCarty	222-222-3222
3	Emily Rayner	333-333-3333

---

### Table 2: Client

---

Client No	Client Name	Property No	Address	Rent Duration	Rent
C1	James Mathew	P1	Electronic City, Mysore	01-Jan-13 to 09-Sep-13	1000
C1	James Mathew	P2	Sankranthi Circle, Mysore	01-Sep-13 to 01-Dec-14	800
C2	Arvind Gupta	P8	Kormangla, Bangalore	01-Dec-13 to 01-Jan-15	1500

---

This table seems to be in 1NF as each column contains atomic values.

### Table 3: Owner

---

Owner No	Owner Name
O101	Tina Joseph
O109	Dinesh Shah
O105	Ram

---

This table is also in 1NF.

Now, the tables are in 1NF, and the data has been appropriately normalized by removing repeating groups or arrays of values.

Explain the term functional dependency with example.

Let R be a relation schema R(A, B, C, D, E)

A B C D E

a 2 3 4 5

b a 3 4 5

c 2 3 6 5

d 2 3 6 6

Check the following functional dependency on the relation R

a.  $A \rightarrow B$

b.  $A \rightarrow CD$

c.  $A \rightarrow BCDE$

d.  $A \rightarrow C$

e.  $BD \rightarrow C$

f.  $ABDE \rightarrow C$

g.  $C \rightarrow DE$

Ans:

In database theory, a functional dependency is a relationship between two sets of attributes in a relation (table) such that, for every unique combination of values in one set, there is only one corresponding combination of values in the other set. In other words, it describes how the values of one set of attributes determine the values of another set.

Let's consider the relation R(A, B, C, D, E) with the given data:

...

A B C D E

a 2 3 4 5

b a 3 4 5

c 2 3 6 5

d 2 3 6 6

...

Now, let's check the functional dependencies:

a.  $A \rightarrow B$ : A determines B if, for every unique value of A, there is only one corresponding value of B. In this case, it is not true because for  $A=a$ , there are two values of B (2 and a). Therefore, A does not determine B.

b.  $A \rightarrow CD$ : A determines CD if, for every unique value of A, there is only one corresponding combination of values for CD. In this case, it is not true because for  $A=a$ , there are two combinations of CD (3 4 and 3 4). Therefore, A does not determine CD.

c.  $A \rightarrow BCDE$ : A determines BCDE if, for every unique value of A, there is only one corresponding combination of values for BCDE. In this case, it is not true because for  $A=a$ , there are two combinations of BCDE (2 3 4 5 and a 3 4 5). Therefore, A does not determine BCDE.

d.  $A \rightarrow C$ : A determines C if, for every unique value of A, there is only one corresponding value of C. In this case, it is true because for each value of A, there is only one corresponding value of C.

e.  $BD \rightarrow C$ : BD determines C if, for every unique combination of values for BD, there is only one corresponding value of C. In this case, it is not true because for  $BD=(2\ 3)$ , there are two values of C (4 and 6). Therefore, BD does not determine C.

f.  $ABDE \rightarrow C$ : ABDE determines C if, for every unique combination of values for ABDE, there is only one corresponding value of C. In this case, it is not true because for  $ABDE=(a\ 2\ 5\ 5)$ , there are two values of C (4 and 5). Therefore, ABDE does not determine C.

g.  $C \rightarrow DE$ : C determines DE if, for every unique value of C, there is only one corresponding combination of values for DE. In this case, it is not true because for  $C=3$ , there are two combinations of DE (4 5 and 6 5). Therefore, C does not determine DE.

8. Find the highest normal form of a relation.

1)  $R(A,B,C,D,E)$  with FD set  $\{A \rightarrow D, B \rightarrow A, BC \rightarrow D, AC \rightarrow BE\}$

2)  $R(A,B,C,D,E)$  with FD set as  $\{BC \rightarrow D, AC \rightarrow BE, B \rightarrow E\}$

Ans;

To find the highest normal form of a relation, we need to eliminate any partial, transitive, or other undesirable dependencies. Let's analyze each given relation and its functional dependencies (FDs).

### 1)  $R(A, B, C, D, E)$  with FD set  $\{A \rightarrow D, B \rightarrow A, BC \rightarrow D, AC \rightarrow BE\}$

**\*\*Candidate Keys:\*\***

- A and B are candidate keys because:

- A determines D.
- B determines A, and A determines D (transitive dependency).

**\*\*Analysis:\*\***

- A determines D ( $A \rightarrow D$ ).
- B determines A ( $B \rightarrow A$ ).
- BC determines D ( $BC \rightarrow D$ ).
- AC determines BE ( $AC \rightarrow BE$ ).

**\*\*Normalization Steps:\*\***

1. Decompose  $BC \rightarrow D$  (2NF violation) to  $BC \rightarrow C$  and  $C \rightarrow D$ .
2. Decompose  $AC \rightarrow BE$  (3NF violation) to  $AC \rightarrow B$  and  $AC \rightarrow E$ .

**\*\*Resulting Relations:\*\***

1.  $R_1(A, B, C)$
2.  $R_2(C, D)$
3.  $R_3(A, C, B, E)$

The relations are now in 3NF.

### 2)  $R(A, B, C, D, E)$  with FD set as  $\{BC \rightarrow D, AC \rightarrow BE, B \rightarrow E\}$

**\*\*Candidate Keys:\*\***

- BC and AC are candidate keys because:
  - BC determines D.
  - AC determines BE.

**\*\*Analysis:\*\***

- BC determines D ( $BC \rightarrow D$ ).
- AC determines BE ( $AC \rightarrow BE$ ).
- B determines E ( $B \rightarrow E$ ).

**\*\*Normalization Steps:\*\***

1. Decompose  $BC \rightarrow D$  (2NF violation) to  $BC \rightarrow C$  and  $C \rightarrow D$ .
2. Decompose  $AC \rightarrow BE$  (3NF violation) to  $AC \rightarrow B$  and  $AC \rightarrow E$ .

**\*\*Resulting Relations:\*\***

1.  $R_1(A, B, C)$
2.  $R_2(C, D)$
3.  $R_3(A, C, B, E)$

The relations are now in 3NF.

In both cases, the highest normal form achieved is 3NF (Third Normal Form).

9. Explain how functional dependencies can be used to indicate the following:

A one-to-one relationship set exists between entity sets student and instructor.

A many-to-one relationship set exists between entity sets student and instructor.

Ans:

Functional dependencies play a crucial role in database design and normalization, helping to define relationships between attributes in different tables. Let's discuss how functional dependencies can indicate the specified relationships:

#### ### 1. One-to-One Relationship Set:

In a one-to-one relationship set between entity sets `student` and `instructor`, each student is associated with at most one instructor, and each instructor is associated with at most one student. This can be represented using functional dependencies.

Let's assume we have the following attributes for the entity sets `student` and `instructor`:

- Attributes of the `student` entity set: {student\_id, student\_name, ...}
- Attributes of the `instructor` entity set: {instructor\_id, instructor\_name, ...}

To indicate a one-to-one relationship set, we can define functional dependencies as follows:

```
```plaintext
{student_id} → {instructor_id}
{instructor_id} → {student_id}
```
```

In these functional dependencies, each student is associated with exactly one instructor (indicated by  $\{student\_id\} \rightarrow \{instructor\_id\}$ ), and each instructor is associated with exactly one student (indicated by  $\{instructor\_id\} \rightarrow \{student\_id\}$ ). Together, these dependencies imply a one-to-one relationship set between `student` and `instructor`.

#### ### 2. Many-to-One Relationship Set:

In a many-to-one relationship set between entity sets `student` and `instructor`, each student can be associated with at most one instructor, but each instructor can be associated with multiple students. Again, we use functional dependencies to represent this.

Functional dependencies:

```
```plaintext
{student_id} → {instructor_id}
```
```

In this case, each student is associated with exactly one instructor, but an instructor may be associated with multiple students. The functional dependency  $\{student\_id\} \rightarrow \{instructor\_id\}$  indicates the many-to-one relationship set between the `student` and `instructor` entity sets.

It's important to note that the functional dependencies discussed here are conceptual and do not necessarily translate directly into a specific relational schema. The actual implementation may involve the use of foreign keys, primary keys, and other constraints based on the database design principles.

10. Compute the closure of the following set  $F$  of functional dependencies for relation schema  $R = (A, B, C, D, E)$ .

$A \rightarrow BC$

$CD \rightarrow E$

$B \rightarrow D$

$E \rightarrow A$

List the candidate keys for  $R$ .

Ans:

To compute the closure of a set of functional dependencies, you can use the Armstrong's axioms. The closure of a set of functional dependencies  $(F^+)$  is the set of all functional dependencies that can be logically inferred from  $(F)$  using Armstrong's axioms.

The closure of a set of attributes  $(X)$  with respect to a set of functional dependencies  $(F)$ , denoted as  $(X^+)$ , is the set of all attributes that are functionally determined by  $(X)$  under  $(F)$ .

Let's compute the closure of the given set of functional dependencies  $(F)$  for the relation schema  $(R = (A, B, C, D, E))$ :

1.  **$A \rightarrow BC$** :

- $(A^+ = A)$  (trivially)
- $((A, B)^+ = (A, B, C))$  (using  $(A \rightarrow BC)$ )

2.  **$CD \rightarrow E$** :

- $(CD^+ = CD)$  (trivially)
- $((CD)^+ = (CD, E))$  (using  $(CD \rightarrow E)$ )

3.  **$B \rightarrow D$** :

- $(B^+ = B)$  (trivially)
- $((B)^+ = (B, D))$  (using  $(B \rightarrow D)$ )

4.  **$E \rightarrow A$** :

- $(E^+ = E)$  (trivially)
- $((E)^+ = (E, A))$  (using  $(E \rightarrow A)$ )

Now, combine all the results:

$F^+ = \{A, B, C, D, E, AB, AC, BC, CD, E, EAB, EAC, EBC, ECD\}$

Next, let's find the candidate keys for  $(R)$ . A candidate key is a minimal superkey for a relation. A superkey is a set of attributes that can uniquely determine all other attributes in the relation.

Start with individual attributes and combine them, checking if the closure of each combination contains all attributes.

Possible candidate keys:

- $(A)$  (already a superkey, as  $(A^+ = \{A, B, C, D, E\})$ )
- $(B)$  (already a superkey, as  $(B^+ = \{B, D\})$ )
- $(C)$  (already a superkey, as  $(C^+ = \{C\})$ )
- $(D)$  (already a superkey, as  $(D^+ = \{D\})$ )
- $(E)$  (already a superkey, as  $(E^+ = \{E, A, B, C, D\})$ )

So, the candidate keys for  $(R)$  are  $(A, B, C, D, E)$ .

1. Define Normalization and illustrate with example First Normal Form, Second Normal Form & Third Normal Form respectively

Ans:

Normalization is a database design technique used to organize data in a relational database efficiently. The goal of normalization is to reduce data redundancy and dependency by

organizing the data into related tables. The process involves breaking down large tables into smaller, related tables and defining relationships between them. There are several normal forms, with the first three being the most commonly discussed: First Normal Form (1NF), Second Normal Form (2NF), and Third Normal Form (3NF).

#### 1. \*\*First Normal Form (1NF):\*\*

- A table is in 1NF if it contains only atomic (indivisible) values and there are no repeating groups or arrays.
- Example: Consider a table with student information where each student can have multiple phone numbers.

| StudentID | Name  | Phone Numbers      |
|-----------|-------|--------------------|
| 1         | Alice | 555-1234, 555-5678 |
| 2         | Bob   | 555-9876           |

This table is not in 1NF because the "Phone Numbers" column contains multiple values. To normalize it, you would create a separate table for phone numbers linked to the student ID.

#### \*\*Normalized Tables:\*\*

- Students Table:

| StudentID | Name  |
|-----------|-------|
| 1         | Alice |
| 2         | Bob   |

- Phone Numbers Table:

| StudentID | Phone Number |
|-----------|--------------|
| 1         | 555-1234     |
| 1         | 555-5678     |
| 2         | 555-9876     |

#### 2. \*\*Second Normal Form (2NF):\*\*

- A table is in 2NF if it is in 1NF and all non-key attributes are fully functionally dependent on the primary key.
- Example: Continuing with the student example, suppose we have a table with courses and the instructors who teach them.

| StudentID | Course | Instructor |
|-----------|--------|------------|
|-----------|--------|------------|



| StudentID | Course    | Instructor  |
|-----------|-----------|-------------|
| 1         | Math      | Dr. Smith   |
| 2         | History   | Dr. Johnson |
| 1         | Chemistry | Dr. White   |

In this table, the combination of "StudentID" and "Course" is the primary key, but "Instructor" depends only on "Course," not the entire key.

**\*\*Normalized Tables:\*\***

- Students Table:

| StudentID | Name  |
|-----------|-------|
| 1         | Alice |
| 2         | Bob   |

- Courses Table:

| Course    | Instructor  |
|-----------|-------------|
| Math      | Dr. Smith   |
| History   | Dr. Johnson |
| Chemistry | Dr. White   |

### 3. **\*\*Third Normal Form (3NF):\*\***

- A table is in 3NF if it is in 2NF, and there are no transitive dependencies. That is, no non-prime attribute depends on another non-prime attribute.

- Example: Building on the previous example, suppose the instructor's office location is also recorded.

| Course    | Instructor  | Office Location |
|-----------|-------------|-----------------|
| Math      | Dr. Smith   | Room 101        |
| History   | Dr. Johnson | Room 202        |
| Chemistry | Dr. White   | Room 303        |

In this table, "Office Location" depends on "Instructor," which is not a primary key.

**\*\*Normalized Tables:\*\***

- Students Table:

| StudentID | Name  |
|-----------|-------|
| 1         | Alice |
| 2         | Bob   |

|   |       |  |
|---|-------|--|
| 1 | Alice |  |
| 2 | Bob   |  |

- Courses Table:

|           |             |  |
|-----------|-------------|--|
| Course    | Instructor  |  |
| -----     | -----       |  |
| Math      | Dr. Smith   |  |
| History   | Dr. Johnson |  |
| Chemistry | Dr. White   |  |

- Instructors Table:

|             |                 |  |
|-------------|-----------------|--|
| Instructor  | Office Location |  |
| -----       | -----           |  |
| Dr. Smith   | Room 101        |  |
| Dr. Johnson | Room 202        |  |
| Dr. White   | Room 303        |  |

By normalizing the data, we reduce redundancy, improve data integrity, and make the database structure more efficient.

2. Define Normalization and illustrate with example Fourth Normal Form (4NF) & BCNF.

Ans:

Normalization is a process in database design that involves organizing data in a relational database to reduce redundancy and dependency. The goal is to improve data integrity and minimize the chances of data anomalies (such as update, insert, and delete anomalies).

Normalization is typically achieved through a series of normal forms, each building on the previous one.

Fourth Normal Form (4NF) and Boyce-Codd Normal Form (BCNF) are advanced stages of normalization. Let's define each and provide an example for better understanding:

### Fourth Normal Form (4NF):

Fourth Normal Form deals with multi-valued dependencies in a database. A table is in 4NF if it is already in Boyce-Codd Normal Form (BCNF) and there are no non-trivial multi-valued dependencies.

**\*\*Example:\*\***

Consider a table called `Student\_Courses`:

| Student_ID | Course  | Instructor |
|------------|---------|------------|
| 1          | Math    | Mr. A      |
| 1          | Physics | Mr. B      |
| 2          | Math    | Mr. A      |
| 2          | Biology | Mr. C      |

In this table, Student\_ID and Course together determine the Instructor. However, it violates 4NF because there are multi-valued dependencies ( $\text{Student\_ID} \twoheadrightarrow \text{Instructor}$ ,  $\text{Course} \twoheadrightarrow \text{Instructor}$ ). To bring it to 4NF, you would split it into two tables:

**\*\*Student\_Courses:\*\***

| Student_ID | Course  |
|------------|---------|
| 1          | Math    |
| 1          | Physics |
| 2          | Math    |
| 2          | Biology |

**\*\*Course\_Instructors:\*\***

| Course  | Instructor |
|---------|------------|
| Math    | Mr. A      |
| Physics | Mr. B      |
| Biology | Mr. C      |

Now, each table is in 4NF.

**### Boyce-Codd Normal Form (BCNF):**

BCNF is a higher normal form than the Third Normal Form (3NF). A table is in BCNF if, for every non-trivial functional dependency, the determinant is a superkey.

**\*\*Example:\*\***

Consider a table called `Employees`:

| Employee_ID | Project_ID | Employee_Name | Project_Name |
|-------------|------------|---------------|--------------|
|-------------|------------|---------------|--------------|

| Employee_ID | Project_ID | Employee_Name | Project_Name |
|-------------|------------|---------------|--------------|
| 1           | 101        | Alice         | Project_A    |
| 2           | 101        | Bob           | Project_A    |
| 3           | 102        | Alice         | Project_B    |

Here, {Employee\_ID, Project\_ID} is a composite primary key, and there's a functional dependency Employee\_ID, Project\_ID → Employee\_Name, Project\_Name. However, it's not in BCNF because Employee\_ID, Project\_ID is a superkey. To bring it to BCNF, we split the table into two:

**\*\*Employees:\*\***

| Employee_ID | Project_ID | Employee_Name |
|-------------|------------|---------------|
| 1           | 101        | Alice         |
| 2           | 101        | Bob           |
| 3           | 102        | Alice         |

**\*\*Projects:\*\***

| Project_ID | Project_Name |
|------------|--------------|
| 101        | Project_A    |
| 102        | Project_B    |

Now, each table is in BCNF.

## Unit V – Indexing and Hashing

1. Define B+ tree and illustrate with examples and also illustrate how the insertion will take place.

Ans: A B+ tree is a type of self-balancing tree data structure that maintains sorted data and allows searches, insertions, deletions, and sequential access in logarithmic time. B+ trees are commonly used in databases and file systems because of their ability to efficiently support range queries.

Here are the key characteristics of a B+ tree:

1. **\*\*Balanced Structure:\*\*** A B+ tree is balanced, meaning that the height of the tree is always kept minimal. This ensures that the operations on the tree have a logarithmic time complexity.

2. **\*\*Sorted Data:\*\*** The data is stored in the leaf nodes in a sorted order. Each leaf node contains a range of values and a pointer to the next leaf node, forming a linked list for efficient sequential access.

3. **\*\*Non-Leaf Nodes:\*\*** Non-leaf nodes in a B+ tree contain keys and pointers to child nodes. The keys in a non-leaf node are used to guide the search for a particular value.

Now, let's illustrate with an example and show how insertion works:

### B+ Tree Example:

Consider a B+ tree with an order of 3, which means each node can have at most 3 keys.

#### Initial Empty Tree:

...

[]

...

#### Insert 10:

...

[10]

...

#### Insert 5:

...

[5] [10]

...

#### Insert 20:

...

[5] [10, 20]

...

#### Insert 6:

...

[5, 6] [10, 20]

...

#### Insert 12:

```
...
```

```
    [5, 6]    [10, 12, 20]
```

```
...
```

```
##### Insert 3:
```

```
...
```

```
    [3, 5, 6]    [10, 12, 20]
```

```
...
```

```
##### Insert 30:
```

```
...
```

```
    [3, 5, 6]    [10, 12, 20, 30]
```

```
...
```

```
##### Insert 7:
```

```
...
```

```
    [3, 5, 6, 7]    [10, 12, 20, 30]
```

```
...
```

2. Enlist different indexing techniques and illustrate with appropriate examples.

Ans: In database engineering, indexing is a technique used to improve the speed of data retrieval operations on a database. Here are some common indexing techniques along with examples:

1. **\*\*Single-level Index:\*\***

- A single index structure that provides a mapping between the key values and the location of the corresponding records.

- Example: Consider a table of employees with an index on the "EmployeeID" column.

```
```sql
```

```
CREATE INDEX idx_employee_id ON employees(EmployeeID);
```

```
```
```

2. **\*\*Multilevel Index:\*\***

- Uses multiple index structures to speed up access to the data.

- Example: Creating an index on a composite key in a table.

```
```sql
```

```
CREATE INDEX idx_employee_name_salary ON employees(EmployeeName, Salary);
```

```
```
```

3. **\*\*Clustered Index:\*\***

- Determines the physical order of data rows in a table based on the order of the index.
- Example: Creating a clustered index on the "OrderDate" column in an "Orders" table.

```
```sql
CREATE CLUSTERED INDEX idx_order_date ON orders(OrderDate);
```
```

#### 4. **\*\*Non-Clustered Index:\*\***

- Separates the index structure from the data rows, creating a separate structure for faster retrieval.
- Example: Creating a non-clustered index on the "CustomerID" column in a "Customers" table.

```
```sql
CREATE NONCLUSTERED INDEX idx_customer_id ON customers(CustomerID);
```
```

#### 5. **\*\*Bitmap Index:\*\***

- Uses a bitmap for each unique value in the indexed column, indicating the presence or absence of that value in the data.
- Example: Creating a bitmap index on the "Gender" column in an "Employees" table.

```
```sql
CREATE BITMAP INDEX idx_gender ON employees(Gender);
```
```

#### 6. **\*\*Spatial Index:\*\***

- Optimizes the retrieval of spatial data based on geometric properties.
- Example: Creating a spatial index on the "Location" column in a table storing geographical data.

```
```sql
CREATE SPATIAL INDEX idx_location ON spatial_data(Location);
```
```

#### 7. **\*\*Full-Text Index:\*\***

- Enables efficient searching of large text columns by creating an index on words in the text.
- Example: Creating a full-text index on the "Description" column in a table.

```
```sql
```

```
CREATE FULLTEXT INDEX idx_description ON products(Description);  
^^
```

These examples provide a glimpse into how different indexing techniques can be applied to various scenarios in database engineering to enhance query performance. The choice of the indexing technique depends on the specific requirements and characteristics of the data and queries in a given database system.

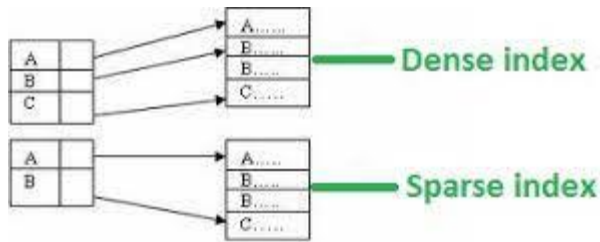
3. Compare between Dense and Sparse Indexing with a suitable diagram.



Ans:

Feature	Dense Indexing	Sparse Indexing
<b>Definition</b>	An index entry for every record in the database table, regardless of whether it contains a null or a non-null value.	An index entry only for records that have non-null values in the indexed column(s). Null values are not explicitly indexed.
<b>Storage Space</b>	Generally requires more storage space because it includes entries for every record in the table.	Requires less storage space compared to dense indexing because it excludes entries for records with null values.
<b>Insertion and Deletion</b>	Insertion and deletion operations can be slower because every insert or delete may require updating the index.	Generally faster for insertion and deletion operations since only non-null values need to be considered for indexing.
<b>Query Performance</b>	Generally provides faster query performance for range queries, as the index covers all records.	May result in slower query performance for range queries since null values are not explicitly indexed, and the query may need to access the actual data for those records.
<b>Suitability</b>	Suitable for columns with low cardinality (few distinct values) where most of the values are non-null.	Suitable for columns with high cardinality (many distinct values) and a significant number of null values.
<b>Example</b>	An index on a column containing status values (e.g., 'active' or 'inactive') where most records have a status value.	An index on a column containing optional attributes, such as comments or additional details, where many records may have null values.

4. Give the difference between ordered indices and Hashing with one example. Which is better to use?



Picture : different database index

Ans:

Criteria	Ordered Indices	Hashing
<b>Ordering</b>	Data is stored in a specific order (ascending or descending) based on the indexed column.	No inherent order; data is distributed randomly.
<b>Search Complexity</b>	$O(\log n)$ - Binary search in ordered index.	$O(1)$ - Constant time complexity in ideal cases.
<b>Range Queries</b>	Efficient for range queries due to ordered storage.	Less efficient for range queries.
<b>Insertion/Deletion</b>	Slower for insertion and deletion as it may require reshuffling of data.	Generally faster for insertion and deletion.
<b>Example</b>	B-tree or B+tree index on a column with a unique constraint.	Hash index on a column with a unique constraint.
<b>Use Cases</b>	Well-suited for columns frequently used in range queries (e.g., timestamps, numerical values).	Effective for equality searches (e.g., primary key lookups) where quick access is crucial.
<b>Space Efficiency</b>	Requires additional space to store the index structure (e.g., B-tree nodes).	Typically more space-efficient as it directly maps values to locations.
<b>Collision Handling</b>	N/A (Collisions may not be a concern as elements are ordered).	Requires a mechanism to handle collisions (e.g., chaining or open addressing).
<b>Sorting</b>	Data is automatically sorted due to the ordered nature of the index.	No inherent sorting, and sorting would require additional effort.

**\*\*Example:\*\***

Consider a database table storing information about students:

```

```plaintext
| StudentID | Name   | Age | GPA |
|-----|-----|-----|-----|
| 101      | Alice  | 20  | 3.8 |
| 102      | Bob    | 22  | 3.5 |
| 103      | Charlie| 21  | 3.9 |
```

```

Now, let's create an index on the "StudentID" column for both ordered indices (B-tree) and hashing.

### 1. **Ordered Indices (B-tree):**

- B-tree index on the "StudentID" column:

```

```
B-tree: [101, 102, 103]
```

```

### 2. **Hashing:**

- Hash index on the "StudentID" column:

```

```
Hash Table:
0: 101
1: 102
2: 103
```

```

### **Which is Better to Use?**

The choice between ordered indices and hashing depends on the specific use case and query patterns. Generally:

- Use ordered indices when range queries are common or when maintaining a sorted order is important.
- Use hashing when quick access for equality searches is crucial, and range queries are not a primary concern.

In many cases, a combination of both techniques is employed in modern database systems to benefit from the strengths of each. For example, PostgreSQL uses a combination of B-tree and hash indices.

5. Define Indexing and illustrate primary Index, Secondary Index and Clustering Index.

Ans: Indexing is a database management system (DBMS) concept used to optimize the retrieval of records from a database table. An index is a data structure that provides a quick and efficient way to look up data based on the values in one or more columns. It enhances the speed of data retrieval operations by creating a logical order of the data.

1. **\*\*Primary Index:\*\***

- A primary index is an index on the primary key of a table. The primary key is a unique identifier for each record in the table.
- The primary index is automatically created when the primary key is defined on a table.
- It ensures that the data is physically organized in the database in the order of the primary key.
- Example: Consider a table of employees with a primary key on the employee ID. The primary index will arrange the data based on employee ID.

2. **\*\*Secondary Index:\*\***

- A secondary index is an index created on a non-primary key column to enhance the retrieval speed for queries that do not use the primary key.
- Unlike the primary index, which determines the physical order of the data, a secondary index provides an alternate way to access the data.
- Example: In the same employee table, a secondary index could be created on the "employee\_name" column to speed up searches based on employee names.

3. **\*\*Clustering Index:\*\***

- A clustering index determines the physical order of the data rows in the table.
- Unlike a regular index, which only provides a logical order for data retrieval, a clustering index directly affects the way data is stored on disk.
- Only one clustering index can be created on a table because it defines the physical order of the data.
- Example: If you have a table of orders with a clustering index on the order date, the data will be physically stored on disk in the order of the order dates.

6. Compare B tree and B+ tree with a suitable example.

| Feature             | B-tree                                                                              | B+ tree                                                                                |
|---------------------|-------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------|
| Structure           | Each node contains both keys and data pointers.                                     | Only leaves contain data pointers; internal nodes contain only keys.                   |
| Search Operation    | Search operations are faster due to fewer levels of indirection.                    | Search operations may require more levels of indirection, which can be slower.         |
| Data Storage        | Data is stored in both internal and leaf nodes.                                     | Data is only stored in leaf nodes.                                                     |
| Node Splitting      | Splits can occur at any level, including internal nodes.                            | Splits only occur at the leaf level; internal nodes contain keys for routing purposes. |
| Range Queries       | May be less efficient for range queries due to the need to traverse internal nodes. | Well-suited for range queries as they involve only leaf nodes.                         |
| Fanout              | Typically has a smaller fanout (number of children per node).<br>↓                  | Generally has a larger fanout, leading to a shallower tree.                            |
| Disk I/O Efficiency | Tends to have more I/O operations due to data in internal nodes.                    | Generally more I/O efficient as only leaf nodes are involved in data storage.          |
| Sequential Access   | Sequential access can be less efficient.                                            | Sequential access is more efficient due to the linked list structure of leaf nodes.    |
| Example             | Used in file systems and some relational databases.                                 | Commonly used in relational databases for indexing.                                    |

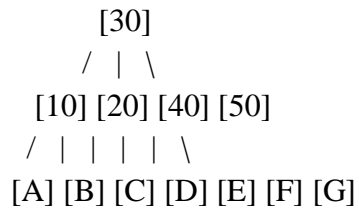
Let's consider an example to illustrate the differences. Suppose we have a B-tree and a B+ tree indexing the following key-value pairs:

(10, A), (20, B), (30, C), (40, D), (50, E), (60, F), (70, G)

...

### B-tree:

...

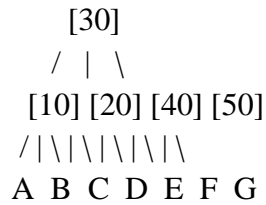


...

In a B-tree, data is stored in both internal and leaf nodes.

### B+ tree:

...



...

In a B+ tree, data is only stored in the leaf nodes, and internal nodes contain keys for routing purposes. This structure allows for more efficient range queries and sequential access.

7. Compare indexing and hashing and illustrate with examples.

| Criteria                  | Indexing                                                                                                                                                                                              | Hashing                                                                                                                                                                                                       |
|---------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Definition</b>         | Indexing is a data structure that improves the speed of data retrieval operations on a database at the cost of additional writes and storage space.                                                   | Hashing is a technique that converts a key into an index to quickly locate the desired data in a table.                                                                                                       |
| <b>Function</b>           | Speeds up data retrieval by creating a sorted structure that allows for efficient search and retrieval.                                                                                               | Speeds up data retrieval by directly calculating the storage location using a hash function.                                                                                                                  |
| <b>Structure Type</b>     | B-trees, B+ trees, and other structures depending on the database management system.                                                                                                                  | Hash tables or hash indexes.                                                                                                                                                                                  |
| <b>Example</b>            | Suppose you have a table of employee records, and you create an index on the "EmployeeID" column. This index will store sorted values of EmployeeID along with pointers to the corresponding records. | Consider a hash table storing email addresses. The hash function might convert the email into an index that corresponds to the location where the email's associated data (e.g., user information) is stored. |
| <b>Search Efficiency</b>  | Generally efficient for range queries and inequality searches due to sorted structure.                                                                                                                | Efficient for equality searches but less efficient for range queries compared to indexing.                                                                                                                    |
| <b>Insertion/Deletion</b> | Inserts and deletes can be slower because the index structure needs to be updated.                                                                                                                    | Inserts and deletes are generally faster since the hash function can be recalculated without affecting the overall structure.                                                                                 |
| <b>Use Cases</b>          | Well-suited for scenarios where there are frequent range queries or inequality searches.                           | Well-suited for scenarios where quick equality searches are predominant, such as searching for a specific record based on a                                                                                   |

8. Define Bitmap Indices and illustrate with suitable example

Ans:Bitmap indices are a type of data structure used in database management systems to optimize the retrieval of data in scenarios where the data has a limited number of distinct values, such as categorical or boolean attributes. Bitmap indices represent sets of values as bitmaps, where each bit corresponds to a unique attribute value, and the presence or absence of a bit indicates whether a particular value is associated with a given record.

Here's a simple example to illustrate bitmap indices in the context of a relational database:

### Example Scenario:

Let's consider a database for a library with the following schema:

- **\*\*Book Table:\*\***
- BookID (Primary Key)
- Title
- Author
- Genre
- Available (Boolean indicating whether the book is available for borrowing)

### Bitmap Index Creation:

Suppose we want to create a bitmap index on the "Genre" attribute. The genres in our database are limited, say "Fiction," "Mystery," "Science Fiction," and "Non-Fiction."

#### Original Data:

| BookID | Title             | Author              | Genre           | Available |
|--------|-------------------|---------------------|-----------------|-----------|
| 1      | The Great Gatsby  | F. Scott Fitzgerald | Fiction         | Yes       |
| 2      | 1984              | George Orwell       | Science Fiction | Yes       |
| 3      | The Da Vinci Code | Dan Brown           | Mystery         | No        |
| 4      | Sapiens           | Yuval Noah Harari   | Non-Fiction     | Yes       |

#### Bitmap Indices:

- **\*\*Fiction:\*\*** 1 0 0 0
- **\*\*Mystery:\*\*** 0 0 1 0
- **\*\*Science Fiction:\*\*** 0 1 0 0
- **\*\*Non-Fiction:\*\*** 0 0 0 1



Each column in the bitmap corresponds to a book, and the rows correspond to the presence or absence of a particular genre for each book. A '1' indicates that the book belongs to that genre, and '0' indicates that it doesn't.

### ### Query Optimization:

Now, suppose we want to find all available fiction books. With the bitmap index, we can perform a bitwise AND operation between the "Fiction" bitmap and the "Available" bitmap to quickly identify the books that meet both criteria.

- **Available:** 1 1 0 1 (1 for available, 0 for not available)

Performing a bitwise AND operation:

```

Fiction Bitmap  1 0 0 0
AND
Available Bitmap 1 1 0 1
-----
Result         1 0 0 0

```

The result indicates that only the book with BookID 1 ("The Great Gatsby") satisfies both conditions.

Bitmap indices are efficient for queries involving multiple conditions, especially when dealing with low cardinality attributes, as they use bitwise operations that can be processed quickly by computers.

9. Define sparse and dense indexing and illustrate the advantages and disadvantages of it.

Ans: Sparse indexing and dense indexing are two different approaches to organizing and accessing data in various data structures, particularly in the context of databases and information retrieval systems.

1. **Sparse Indexing:**

- **Definition:** In sparse indexing, index entries are created only for the non-null or non-zero values in a dataset. It is particularly useful when dealing with datasets that have a large number of empty or zero values.

- **Advantages:**

- **Space Efficiency:** Sparse indexing requires less storage space because it only stores information about non-empty or non-zero values.
- **Reduced Maintenance Overhead:** Since it doesn't index empty or zero values, updates to these values don't require changes to the index structure, reducing maintenance overhead.
- **Disadvantages:**
  - **Slower Access for Some Queries:** Retrieving values associated with empty or zero positions may require scanning the entire dataset, leading to slower access times for certain queries.
  - **Complexity in Implementation:** Implementing sparse indexing can be more complex than dense indexing due to the need to handle variable gaps in the index structure.

## 2. **Dense Indexing:**

- **Definition:** In dense indexing, index entries are created for every value in a dataset, whether it's empty, zero, or non-empty. It is a more straightforward indexing approach.
- **Advantages:**
  - **Faster Access:** Since an index entry exists for every value, access times for queries are generally faster because the index can directly point to the location of the desired value.
  - **Simpler Implementation:** Dense indexing is often simpler to implement compared to sparse indexing, as it doesn't require handling variable gaps in the index structure.
- **Disadvantages:**
  - **Increased Storage Overhead:** Dense indexing can consume more storage space, especially in datasets with a large number of empty or zero values.
  - **Higher Maintenance Overhead:** Updates to the dataset may require updates to the index structure, leading to higher maintenance overhead.

## 10. Define B+ Tree and illustrate the advantages and disadvantages of B+ Tree over B Tree.

A B+ tree and a B tree are both types of self-balancing search trees commonly used in database and file systems to store and retrieve sorted data efficiently. The B+ tree is an extension of the B tree, designed to provide certain advantages over its predecessor.

### ### B Tree:


- **Definition:** A B tree is a self-balancing tree data structure that maintains sorted data and allows for efficient insertion, deletion, and search operations.
- **Advantages:**
  - **Balanced Structure:** B trees are balanced, ensuring relatively uniform access times for all elements.
  - **Efficient for Random Access:** B trees perform well for random access patterns, making them suitable for database systems.
- **Disadvantages:**
  - **Overhead:** B trees may have more non-leaf nodes than B+ trees, leading to increased storage overhead.

- Limited Sequential Access: B trees are less optimal for sequential access due to their structure.

### ### B+ Tree:

- **Definition:** A B+ tree is an extension of the B tree with some modifications to enhance certain aspects, especially when dealing with large datasets and range queries.
- **Advantages:**
  - **Improved Sequential Access:** B+ trees are optimized for sequential access, making them more suitable for range queries and scans.
  - **Reduced Non-Leaf Overhead:** In a B+ tree, non-leaf nodes store only keys, not data pointers, reducing storage overhead.
  - **Better for Disk-Based Systems:** B+ trees are often preferred for disk-based storage systems because sequential access is more efficient.
  - **Support for Duplicate Keys:** B+ trees can handle duplicate keys more efficiently than B trees by linking duplicate keys in the leaf nodes.
- **Disadvantages:**
  - **Increased Height:** B+ trees may have a slightly higher height compared to B trees for the same dataset, potentially affecting search performance.
  - **Complex Maintenance:** Splitting and merging of nodes in B+ trees during insertion and deletion operations can be more complex.

11. Compare static and dynamic hashing with an example.

| Feature                  | Static Hashing                                                                                                                | Dynamic Hashing                                    |
|--------------------------|-------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------|
| Hash Function            | Fixed hash function is used throughout.                                                                                       | Dynamic hash function can change over time.        |
| Bucket Allocation        | Fixed number of buckets created initially.                                                                                    | Number of buckets can dynamically change.          |
| Handling Overflows       | May lead to overflow in a specific bucket.                                                                                    | Overflows are managed using additional structures. |
| Insertions and Deletions | Requires redistribution if load factor changes.                                                                               | Easily accommodates changes in load factor.        |
| Example                  | Consider a static hash table with 10 buckets.                                                                                 | Consider a dynamic hash table with 10 buckets.     |
|                          | Data hashed to bucket based on hash function.                                                                                 | Data dynamically rehashed based on load factor.    |
|                          | If overflow, manual redistribution needed.  | Overflows can be managed using overflow chains.    |
| Static Hashing Example:  | Suppose hash function is $H(x) = x \% 10$ .                                                                                   | Data distribution is initially even.               |
|                          | Data is hashed using this function.                                                                                           | $H(25) = 5$ , $H(42) = 2$ , $H(17) = 7$ , etc.     |
|                          | If overflow in bucket 5, manual redistribution.                                                                               |                                                    |
| Dynamic Hashing Example: | Initially, hash function $H(x) = x \% 10$ .                                                                                   | Data is distributed evenly in the buckets.         |
|                          | As data grows, load factor increases.                                                                                         | When load factor exceeds a threshold,              |
|                          |                                                                                                                               | hash function changes, and buckets are split.      |
|                          |                                                                                                                               | New hash function $H'(x) = x \% 20$ .              |
|                          |                                                                                                                               | Data is rehashed and redistributed accordingly.    |

## Unit VI – Transaction

1. Consider the following transactions and schedule

Transaction 1

UPDATE accounts

SETbalance=balance-100

WHEREacct\_id=31414

Transaction 2

UPDATE accounts

SETbalance=balance\*1.005

ScheduleT:r1(A), r2(A), w2(A), w1(A), r2(B), w2(B)

What will be the values of A and B after schedule T if the initial values are A=200 and B=100?

Ans:Let's analyze the given schedule T and the transactions involved.

Transaction 1:

...

r1(A) // Read A

w1(A) // Write A

...

Transaction 2:

...

r2(A) // Read A

w2(A) // Write A

r2(B) // Read B

w2(B) // Write B

...

Initial values:

...

A = 200

B = 100

...

Now, let's go step by step through the schedule:

1.  $r_1(A)$ : Read A,  $A=200$
2.  $r_2(A)$ : Read A,  $A=200$
3.  $w_2(A)$ : Write A ( $A * 1.005$ ),  $A=200 * 1.005 = 201$
4.  $w_1(A)$ : Write A ( $A - 100$ ),  $A=201 - 100 = 101$
5.  $r_2(B)$ : Read B,  $B=100$
6.  $w_2(B)$ : Write B ( $B * 1.005$ ),  $B=100 * 1.005 = 100.5$

So, after the schedule T, the final values are:

...

$A = 101$

$B = 100.5$

...

2. Check whether the schedule is view serializable or not?

$S : R_2(B); R_2(A); R_1(A); R_3(A); W_1(B); W_2(B); W_3(B);$

Ans: To determine if a schedule is view serializable, we need to check if the final outcome (i.e., the final state of the database) is the same as if the transactions were executed in some serial order that is consistent with their respective orders in the schedule.

Let's analyze the given schedule:

...

$S : R_2(B); R_2(A); R_1(A); R_3(A); W_1(B); W_2(B); W_3(B);$

...

Here,  $r_i(X)$  represents a read operation of variable X by transaction  $T_i$ , and  $w_i(X)$  represents a write operation of variable X by transaction  $T_i$ .

Let's consider a serial order:  $T_1, T_2, T_3$ .

Serial order 1:  $T_1: R_2(B); W_1(B); T_2: R_2(A); W_2(B); T_3: R_1(A); R_3(A); W_3(B);$

In this serial order, the final state would be:

- $T_1$  reads B and writes to B.
- $T_2$  reads A and writes to B.
- $T_3$  reads A, reads B, and writes to B.

The final state is consistent with the original schedule.

Therefore, the given schedule is view serializable.

3. Consider the following schedule for transaction T1, T2 and T3

R1(X)R2(Y)R3(Y)W2(Y)W1(X)W3(X) R2(X) W2(X)

Find out serializable schedule sequence.

Ans: To determine if a schedule is serializable, we can use the precedence graph method. The precedence graph helps us identify whether there are any cycles, which would indicate a non-serializable schedule.

Let's build the precedence graph for the given schedule:

1. R1(X)
2. R2(Y)
3. R3(Y)
4. W2(Y)
5. W1(X)
6. W3(X)
7. R2(X)
8. W2(X)

Now, let's represent the precedence relationships:

- T1: R1(X) → W1(X)
- T2: R2(Y) → W2(Y) → R2(X) → W2(X)
- T3: R3(Y) → W3(X)

Now, let's draw the precedence graph:

```
...
T1:  R1(X)      W1(X)
      \        /
T2:   R2(Y) - W2(Y)
      /        \
T3:  R3(Y)      W3(X)
...
```

In the graph, an edge from  $T_i$  to  $T_j$  means that an operation in  $T_i$  precedes an operation in  $T_j$ .  
Now, check for cycles in the graph:

There are no cycles in the graph, which means the schedule is serializable. Therefore, the serializable schedule sequence is:

$\langle T1, T3, T2 \rangle$

4. Consider the following schedule for transaction  $T1, T2$  and  $T3$

$R1(X)R2(Y)R3(Y)W2(Y)W1(X)W3(X)R2(X)W2(X)$

Find out serializable schedule sequence.

Ans: To determine if a schedule is serializable, we can use the concept of precedence graph. In a precedence graph, each transaction is represented by a node, and there is an edge from  $T_i$  to  $T_j$  if an operation in  $T_i$  precedes an operation in  $T_j$ .

Let's analyze the given schedule and construct the precedence graph:

1.  $\langle R1(X) \rangle$  - No edges.
2.  $\langle R2(Y) \rangle$  - No edges.
3.  $\langle R3(Y) \rangle$  - No edges.
4.  $\langle W2(Y) \rangle$  - No edges.
5.  $\langle W1(X) \rangle$  -  $\langle R1(X) \rangle$  precedes  $\langle W1(X) \rangle$ , so there is an edge from  $T1$  to  $T2$ .
6.  $\langle W3(X) \rangle$  - No edges.
7.  $\langle R2(X) \rangle$  -  $\langle W1(X) \rangle$  precedes  $\langle R2(X) \rangle$ , so there is an edge from  $T2$  to  $T1$ .
8.  $\langle W2(X) \rangle$  -  $\langle R2(Y) \rangle$  precedes  $\langle W2(X) \rangle$ , so there is an edge from  $T2$  to  $T3$ .
9.  $\langle R2(X) \rangle$  - No new edges.
10.  $\langle W2(X) \rangle$  - No new edges.

The precedence graph is as follows:

...

$T1 \text{ -- } W1(X) \text{ ---- } R2(X)$

|  
|

V

$R1(X) \quad W2(X)$

|  
|

V

$T2 \text{ -- } W2(Y) \text{ ---- } R2(Y)$

|  
|



V  
T3 --W3(X)  
...

Now, let's check for cycles in the precedence graph. If there are no cycles, the schedule is serializable.

In this case, there are no cycles in the graph. Therefore, the given schedule is serializable.

One possible serializable schedule is:  $\backslash(T1 \rightarrow T2 \rightarrow T3\backslash)$ .

5. Consider the following schedule:

w1(A), r2(A), w1(B), w3(C), r2(C), r4(B), w2(D), w4(E), r5(D), w5(E)

Draw precedence graph for schedule and test the same for conflict serializability.

To draw the precedence graph for the given schedule and test for conflict serializability, let's analyze the schedule and identify the conflicting operations:

1. \*\*w1(A)\*\*
2. \*\*r2(A)\*\*
3. \*\*w1(B)\*\*
4. \*\*w3(C)\*\*
5. \*\*r2(C)\*\*
6. \*\*r4(B)\*\*
7. \*\*w2(D)\*\*
8. \*\*w4(E)\*\*
9. \*\*r5(D)\*\*
10. \*\*w5(E)\*\*

Now, let's draw the precedence graph based on the above schedule. In a precedence graph, each operation is represented by a node, and directed edges indicate the order of execution.

...

```

w1(A) --> r2(A)
|         |
v         v
w1(B)    r4(B)
|         |
v         |
w3(C) <-- r2(C)
|

```


v  
w2(D) --> r5(D)  
|  
v  
w4(E) --> w5(E)  
...

Now, to test for conflict serializability, we need to check if there are any cycles in the precedence graph. If there are no cycles, the schedule is conflict serializable.

In the above graph, there are no cycles. Therefore, the given schedule is conflict serializable.

It's important to note that this analysis assumes that the transactions involved in the schedule are consistent and follow the standard rules of transactions. If there were conflicting operations within a cycle, it would indicate a conflict, and the schedule would not be conflict serializable.

6. Explain the distinction between the terms serial schedule and serializable schedule

| Criteria               | Serial Schedule                                                                                                                                              | Serializable Schedule                                                                                                                                                      |
|------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Definition</b>      | A schedule in which transactions are executed one after the other without interleaving. Each transaction completes its execution before the next one starts. | A schedule that is equivalent to some serial execution of the transactions, meaning the final state is the same as if the transactions were executed in some serial order. |
| <b>Concurrency</b>     | No concurrency among transactions.                                                                                                                           | Permits some level of concurrency among transactions while maintaining the illusion of serial execution.                                                                   |
| <b>Execution Order</b> | Transactions are executed in a specific order, one after the other.                                                                                          | Transactions can be interleaved and executed in a different order, as long as the final result is equivalent to some serial order.                                         |
| <b>Isolation Level</b> | Highest level of isolation.<br>                                           | Provides varying levels of isolation depending on the concurrency control mechanisms employed.                                                                             |
| <b>Performance</b>     | Typically results in lower performance due to lack of concurrency.                                                                                           | Allows for higher performance by permitting concurrent execution, but may introduce complexities in managing concurrency.                                                  |
| <b>Example</b>         | T1 → T2 → T3                                                                                                                                                 | T1 → T2; T3 or T2 → T1; T3, etc.<br>(various interleaved possibilities as long as the final result is consistent)                                                          |

7. Explain ACID properties.

Ans: ACID is an acronym that stands for Atomicity, Consistency, Isolation, and Durability. These properties are crucial in the context of database transactions, ensuring that database operations are reliable and maintain data integrity. Let's break down each of the ACID properties:

1. **Atomicity:**

- **Definition:** Atomicity ensures that a transaction is treated as a single, indivisible unit of work. Either all the changes made within a transaction are committed to the database, or none of them are.

- **Example:** Consider a banking transaction where money is transferred from one account to another. Atomicity ensures that both the debit from one account and the credit to another account happen as a single, atomic operation. If either operation fails, the entire transaction is rolled back.

## 2. **Consistency:**

- **Definition:** Consistency ensures that a transaction brings the database from one valid state to another. The database must satisfy certain integrity constraints before and after the transaction.

- **Example:** If a database enforces a constraint that all account balances must remain positive, a transaction that attempts to withdraw more money than is available in an account would violate consistency. The system should prevent such transactions or roll them back.

## 3. **Isolation:**

- **Definition:** Isolation ensures that the concurrent execution of transactions does not interfere with the consistency of the database. Each transaction should be executed in isolation from other transactions until it is committed.

- **Example:** Suppose two transactions are running concurrently, each transferring money between different bank accounts. Isolation ensures that the operations within one transaction are not visible to the other until the first transaction is committed. This prevents interference and maintains data integrity.

## 4. **Durability:**

- **Definition:** Durability guarantees that once a transaction is committed, its effects will persist even in the event of a system failure (such as a power outage or crash). The changes made by a committed transaction are permanent.

- **Example:** If a user receives a confirmation message for a successful fund transfer, the database must ensure that this information is not lost, even if the system crashes immediately after the confirmation.

In summary, the ACID properties are fundamental principles in database design and management. They provide a framework for ensuring the reliability, consistency, and durability of transactions, even in the face of system failures or concurrent operations. These properties are essential for maintaining data integrity in various applications, ranging from financial systems to e-commerce platforms.

## 8. Explain view serializability with examples.

Ans: View serializability is a concept in database theory that ensures the consistency of transactions in a database system. It defines a property where the result of executing a set of transactions in parallel is equivalent to some serial execution of those transactions. In other

words, even though transactions may be executed concurrently, the final state of the database should be the same as if the transactions were executed one after the other in some order.

To understand view serializability, let's consider an example with two transactions: T1 and T2.

Let's assume we have a simple banking database with two accounts, A and B, and the following transactions:

1. \*\*T1: Transfer \$50 from A to B\*\*
2. \*\*T2: Transfer \$30 from B to A\*\*

Now, let's consider two schedules (concurrent executions of these transactions):

**\*\*Schedule 1:\*\***

- T1 starts
- T2 starts
- T1 transfers \$50 from A to B
- T2 transfers \$30 from B to A
- T1 completes
- T2 completes

**\*\*Schedule 2:\*\***

- T2 starts
- T1 starts
- T2 transfers \$30 from B to A
- T1 transfers \$50 from A to B
- T2 completes
- T1 completes

In both Schedule 1 and Schedule 2, the individual transactions T1 and T2 are executed in different orders. However, if you observe the final state of the database after each schedule, you will notice that the result is the same: \$50 is transferred from A to B, and \$30 is transferred from B to A.

These schedules are said to be view serializable because they are equivalent in terms of the final state of the database, even though the transactions are interleaved in different ways.

In a view serializable schedule, the transactions can be reordered while still maintaining the consistency of the final database state. The goal is to ensure that the result is as if the transactions

were executed in some sequential order, preserving the consistency of the database despite concurrent execution.

9. Explain conflict serializability with examples.

Ans: Conflict serializability is another concept in database theory that focuses on the order of conflicting operations between transactions. Conflicting operations are those that access the same data item, where at least one of them is a write operation. A schedule is considered conflict serializable if the order of conflicting operations in the schedule is equivalent to the order in some serial schedule.

Let's consider an example with two transactions: T1 and T2.

1. \*\*T1: Read A; Write A\*\*
2. \*\*T2: Read A; Write A\*\*

Now, let's examine two different schedules:

**\*\*Schedule 1:\*\***

- T1 starts
- T2 starts
- T1 reads A
- T2 reads A
- T1 writes A
- T2 writes A
- T1 completes
- T2 completes

**\*\*Schedule 2:\*\***

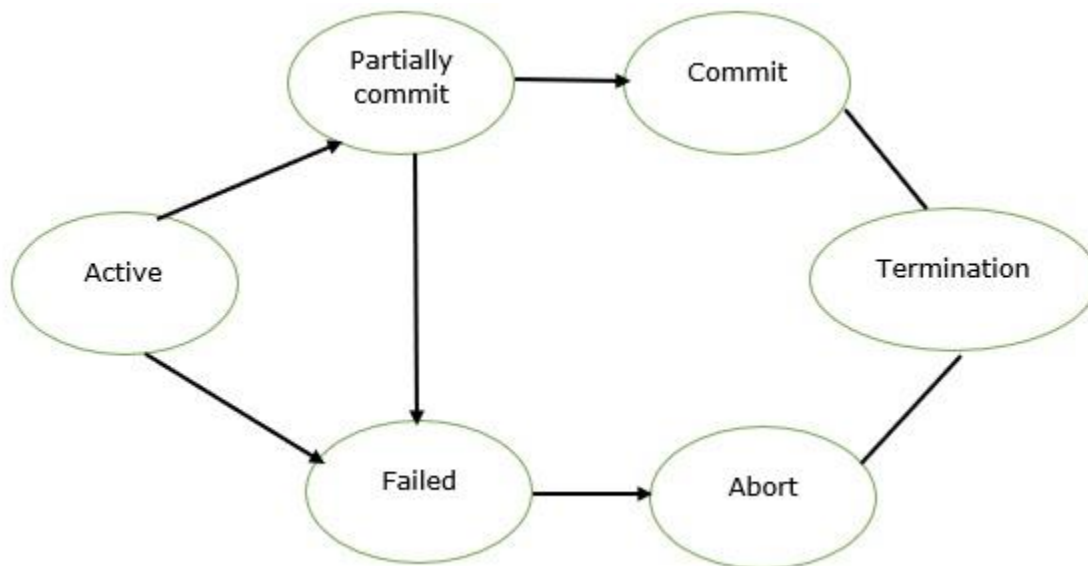
- T2 starts
- T1 starts
- T2 reads A
- T1 reads A
- T2 writes A
- T1 writes A
- T2 completes
- T1 completes

In both Schedule 1 and Schedule 2, the transactions T1 and T2 are interleaved in different orders. However, if you focus on the conflicting operations (the write operations on A), you'll notice that the order of these conflicting operations is the same in both schedules.

Therefore, these schedules are conflict serializable because the order of conflicting operations is equivalent. In this case, the conflicting operations are the writes to A, and the schedules are conflict equivalent despite the interleaving of the read and write operations.

The goal of conflict serializability is to ensure that the final state of the database, considering only the conflicting operations, is equivalent to the final state in some serial execution of the transactions. This helps maintain consistency in the presence of concurrent transactions that operate on the same data items.

10. Explain transaction states with diagrams.



In a database system, a transaction goes through different states during its lifecycle. The standard states are **Active**, **Partially Committed**, **Committed**, and **Failed**. Here's an explanation of each state along with diagrams to illustrate the transitions:

1. **Active (A):**

- In this state, a transaction is actively executing its operations.
- The transaction is performing read and write operations on the database.
- The transaction stays in the active state until it either completes its operations successfully or encounters an error.

2. **Partially Committed (PC):**

- When a transaction completes its execution successfully, it enters the Partially Committed state.
- In this state, the transaction has finished its operations, but the changes it made are not yet permanent.

- The system ensures that all participating transactions are ready to commit before allowing any of them to commit.

3. **\*\*Committed (C):\*\***

- Once a transaction is allowed to commit, it enters the Committed state.
- In this state, the changes made by the transaction become permanent, and the transaction is considered successful.
- Other transactions can now see the committed changes.

4. **\*\*Failed (F):\*\***

- If a transaction encounters an error during its execution, it enters the Failed state.
- In this state, the changes made by the transaction are rolled back, and the database returns to its state before the transaction starts.
- A failed transaction might be retried or abandoned, depending on the application's error-handling strategy.

## **Unit VII – Concurrency Control**

1. Explain two phase locking protocol with strict and rigorous 2PL with example.

The Two-Phase Locking (2PL) protocol is a concurrency control mechanism used in database management systems to ensure consistency and avoid conflicts between transactions. It consists of two phases: the growing phase and the shrinking phase.

1. **\*\*Growing Phase:\*\***

- In this phase, a transaction can acquire locks but cannot release any.
- The transaction can request and acquire locks on data items it needs to access.
- Once a transaction releases a lock, it cannot request any more locks.

2. **\*\*Shrinking Phase:\*\***

- After a transaction releases its first lock, it enters the shrinking phase.
- In this phase, the transaction can release locks but cannot acquire any new locks.
- Once a transaction releases all its locks, it completes, and no further locks can be acquired.

There are two variations of the 2PL protocol: strict 2PL and rigorous 2PL.

**\*\*Strict 2PL:\*\***

- In strict 2PL, a transaction holds all its locks until it reaches the commit point.
- No lock is released until the transaction is ready to commit.
- This ensures that no other transaction can access the locked data until the current transaction completes.



### **\*\*Rigorous 2PL:\*\***

- Rigorous 2PL is a more flexible version of 2PL.
- Unlike strict 2PL, rigorous 2PL allows a transaction to release its locks before reaching the commit point.
- However, once a transaction releases a lock, it cannot acquire any new locks.
- This allows other transactions to access the released data while ensuring that the current transaction does not acquire new locks.

### **\*\*Example:\*\***

Consider two transactions, T1 and T2, accessing two data items, A and B.

#### 1. **\*\*Strict 2PL:\*\***

- T1 acquires a lock on A.
- T2 requests a lock on A but has to wait since T1 holds the lock.
- T1 also acquires a lock on B.
- T2 still waits for A.
- T1 completes and releases both locks.
- Now, T2 can acquire a lock on A and proceed.

#### 2. **\*\*Rigorous 2PL:\*\***

- T1 acquires a lock on A.
- T2 requests a lock on A but has to wait.
- T1 releases the lock on A.
- T2 acquires a lock on A and proceeds.
- T1 acquires a lock on B.
- T2 completes and releases the lock on A.
- T1 proceeds and releases the lock on B.

In strict 2PL, T2 would have to wait until T1 releases all its locks. In rigorous 2PL, T2 can acquire the lock on A once it is released, allowing for more concurrency.

#### 2. Explain lock based protocol with the help of an example.

A lock-based protocol is a concurrency control mechanism used in database management systems to ensure that transactions are executed in a controlled and synchronized manner to maintain data consistency. The basic idea is to use locks to control access to shared resources, such as database records or tables, to prevent conflicts and maintain the integrity of the data.

Here's a simple explanation of the lock-based protocol using an example:

Let's consider a scenario where two transactions, T1 and T2, are trying to update a shared resource (e.g., a bank account balance) concurrently.

1. **\*\*Shared Resource: Bank Account Balance\*\***

Let's say we have a bank account with a current balance of \$1,000.

2. **\*\*Transaction T1:\*\***

- T1 wants to transfer \$200 from the account.
- Before performing the update, T1 requests a lock on the bank account.
- The system grants T1 the lock, and T1 updates the account balance to \$800.
- After completing the update, T1 releases the lock, allowing other transactions to access the account.

3. **\*\*Transaction T2:\*\***

- Simultaneously, T2 also wants to transfer \$300 from the account.
- T2 requests a lock on the bank account before performing the update.
- However, since T1 already holds the lock, T2 is forced to wait until the lock is released by T1.
- Once T1 completes its transaction and releases the lock, T2 is granted the lock.
- T2 updates the account balance to \$500 and then releases the lock.

By using locks, the system ensures that only one transaction can access and modify the shared resource at a time. This prevents conflicts and ensures data consistency. In this example, if both T1 and T2 were allowed to update the account balance simultaneously without locks, the final balance could be incorrect, and the system would lose data consistency.

There are different types of locks, such as shared locks and exclusive locks, and various lock-based protocols, including two-phase locking, to handle different scenarios and requirements in a multi-transaction environment.

3. Explain the implementation of locking with the help of an example.

Locking is a mechanism used in computer science to control access to shared resources, preventing multiple threads or processes from accessing the resource simultaneously. This is crucial in concurrent programming to avoid data corruption or inconsistent states caused by multiple entities attempting to modify shared data concurrently.

Let's discuss a simple example in Python using the `threading` module to illustrate the concept of locking:

```
```python
```

```

import threading

# Shared resource
shared_variable = 0

# Creating a lock
lock = threading.Lock()

# Function to increment the shared variable
def increment_variable():
    global shared_variable

    # Acquiring the lock before modifying the shared variable
    lock.acquire()
    try:
        # Critical section: accessing/modifying the shared resource
        shared_variable += 1
        print(f'Shared variable after increment: {shared_variable}')
    finally:
        # Releasing the lock to allow other threads to acquire it
        lock.release()

# Creating two threads that increment the shared variable
thread1 = threading.Thread(target=increment_variable)
thread2 = threading.Thread(target=increment_variable)

# Start both threads
thread1.start()
thread2.start()

# Wait for both threads to finish
thread1.join()
thread2.join()

```

In this example, we have a shared variable (`shared\_variable`) that multiple threads (`thread1` and `thread2`) will attempt to increment concurrently. To prevent race conditions and ensure the integrity of the shared resource, we use a lock (`lock`).

Here's a breakdown of the code:

1. Import the `threading` module.
2. Create a shared variable (`shared_variable`) and a lock (`lock`).
3. Define a function (`increment_variable`) that increments the shared variable. Inside the function, acquire the lock before modifying the shared resource and release the lock afterward. The critical section is the part of the code where the shared resource is accessed or modified.
4. Create two threads (`thread1` and `thread2`) that call the `increment_variable` function.
5. Start both threads using the `start` method.
6. Wait for both threads to finish using the `join` method.

By using a lock, we ensure that only one thread can enter the critical section (modify the shared variable) at a time. This prevents race conditions and guarantees the integrity of the shared resource.

4. Explain graph-based protocol with the help of examples along with advantages and disadvantages.

A graph-based protocol is a type of communication or interaction system that relies on the representation of data and relationships between entities using a graph structure. In this context, a graph consists of nodes (representing entities) and edges (representing relationships or connections between entities). This approach is often used in various domains, such as computer science, communication networks, social networks, and more.

#### **\*\*Example: Social Network\*\***

Let's consider a social network as an example of a graph-based protocol. In a social network, individuals are represented as nodes, and the connections between individuals (friendship, follows, etc.) are represented as edges.

- **\*\*Nodes:\*\*** Individual user profiles
- **\*\*Edges:\*\*** Friendships or follows between users

#### **Advantages of a Graph-Based Protocol:**

1. **\*\*Flexibility:\*\*** Graphs can represent a wide range of relationships and structures, making them adaptable to different scenarios.
2. **\*\*Efficient Querying:\*\*** Graph databases excel in querying relationships, enabling efficient retrieval of connected data.

3. **Pattern Recognition:** Graphs facilitate the identification of patterns, anomalies, and clusters in data due to their inherent visual nature.

4. **Scalability:** Graph databases can scale horizontally by adding more nodes to handle growing amounts of data and relationships.

#### Disadvantages of a Graph-Based Protocol:

1. **Complexity:** Managing and navigating large graphs can become complex, especially when dealing with intricate relationships.

2. **Performance Challenges:** While graphs excel in certain types of queries, they might not perform as well for other types of queries, depending on the use case.

3. **Storage Overhead:** Storing relationships explicitly can lead to increased storage requirements compared to other data structures.

4. **Limited Use Cases:** While powerful for certain scenarios (e.g., social networks, recommendation systems), graphs may not be the optimal choice for all types of data and applications.

5. Explain deadlock detection and recovery.

Deadlock detection and recovery are techniques used in computer systems, particularly in operating systems, to manage and mitigate the problem of deadlocks. A deadlock occurs when two or more processes are unable to proceed because each is waiting for the other to release a resource. Deadlocks can lead to system inefficiency and unresponsiveness. To address this issue, systems employ deadlock detection and recovery mechanisms.

#### **Deadlock Detection:**

1. **Resource Allocation Graph (RAG):** One common method for deadlock detection is to use a resource allocation graph (RAG). In this graph, nodes represent processes and resources, and edges represent resource requests and allocations.

2. **Wait-Die and Wound-Wait Schemes:** These are used in transaction processing systems. In the Wait-Die scheme, older transactions wait for younger ones to release resources, while in the Wound-Wait scheme, younger transactions wait for older ones.

3. **Timeouts and Probing:** Systems may employ timeouts to periodically check for potential deadlocks. If a process or transaction is taking too long to complete, the system may assume a deadlock and take corrective action.

#### **Deadlock Recovery:**

1. **Process Termination:** One way to recover from a deadlock is to terminate one or more processes involved in the deadlock. The system can choose which processes to terminate based on priority, age, or other criteria. However, this approach may lead to loss of work and is not always feasible, especially if the terminated process has made significant progress.

2. **Resource Preemption:** Instead of terminating processes, the system can preemptively release resources from one or more processes to break the deadlock. The freed resources can then be allocated to other waiting processes. Care must be taken to ensure that preemption does not lead to resource starvation.

3. **Rollback and Restart:** In transaction processing systems, a technique involves rolling back the transactions involved in the deadlock to a previous consistent state and then restarting them. This ensures that the system can make progress without the need for process termination.

4. **Dynamic Resource Allocation:** Some systems dynamically allocate resources to prevent deadlocks. This involves temporarily granting resources to a process and taking them back when the process has finished using them. This method requires careful monitoring and control to avoid resource contention.

It's important to note that deadlock detection and recovery mechanisms add some overhead to the system, and the choice of method depends on the system's requirements, characteristics, and the nature of the applications running on it.

6. Explain time-stamp ordering protocol.

The term "time-stamp ordering protocol" is commonly associated with distributed databases and distributed systems. It is a technique used to establish a total order of transactions in a distributed environment based on the timestamps assigned to each transaction. The goal is to ensure consistency and coordination among the nodes in the system.

Here's an explanation of the time-stamp ordering protocol:

1. **Timestamp Assignment:**

- Each transaction that enters the system is assigned a globally unique timestamp. This timestamp reflects the order in which the transaction was initiated or received by the system.

Commonly, timestamps are assigned using a monotonically increasing counter or a combination of a unique identifier and a logical clock.

2. **\*\*Transaction Execution:\*\***

- Transactions are executed on different nodes of the distributed system. Each node maintains its local clock, and the transactions are processed in the order of their timestamps.

3. **\*\*Message Exchange:\*\***

- To maintain consistency and order, nodes in the distributed system communicate with each other to exchange information about the transactions they have processed. This communication ensures that nodes are aware of the order in which transactions are executed.

4. **\*\*Comparison and Ordering:\*\***

- When nodes receive information about transactions executed at other nodes, they compare the timestamps of these transactions. The time-stamp ordering protocol ensures that transactions are ordered based on their timestamps. If a transaction T1 has a lower timestamp than another transaction T2, then T1 is considered to have occurred before T2.

5. **\*\*Transaction Commit:\*\***

- Nodes can now commit the transactions in the globally agreed-upon order. The commitment of a transaction implies that its effects are permanent and visible to other transactions in the system.

6. **\*\*Concurrency Control:\*\***

- The time-stamp ordering protocol also plays a crucial role in enforcing concurrency control. It helps prevent conflicts and ensures that transactions are executed in a consistent and coordinated manner across the distributed system.

7. **\*\*Handling Concurrent Transactions:\*\***

- In cases where two transactions have the same timestamp or their timestamps are very close, additional mechanisms may be employed to handle concurrency, such as using the transaction's unique identifier as a tiebreaker.

By using timestamp ordering, distributed systems can achieve a total order of transactions, providing a basis for maintaining consistency and coherence across multiple nodes. This is crucial for ensuring the correctness and reliability of distributed databases and systems.

7. Explain validation based protocol with examples.

In a DBMS, validation typically refers to ensuring that data entered into the database meets certain criteria or rules. This is often done to maintain data integrity and consistency. There are two main types of validation in the context of DBMS: data validation and transaction validation.

1. **Data Validation:**

- **Example:** Consider a database for an online bookstore. You might have a rule that ensures the ISBN (International Standard Book Number) entered for a new book is unique. This is a form of data validation to maintain the uniqueness of book identifiers.

2. **Transaction Validation:**

- **Example:** In a banking application, when a customer transfers money from one account to another, the system needs to validate whether the source account has sufficient funds to complete the transaction. If the validation fails, the entire transaction is rolled back, ensuring that the database remains in a consistent state.

8. Explains lock types for data with example.

Sure, there are various lock types used in computer science and databases to control access to shared data. Here are some common lock types with brief explanations and examples:

1. **Shared Lock (S-lock):**

- **Purpose:** Allows multiple transactions to read a resource simultaneously but prevents any of them from writing.

- **Example:** Several users can read the contents of a file simultaneously, but none of them can modify it until all readers have released their shared locks.

2. **Exclusive Lock (X-lock):**

- **Purpose:** Grants exclusive access to a resource, preventing other transactions from both reading and writing.

- **Example:** A database transaction that is updating a record may acquire an exclusive lock on that record, preventing other transactions from reading or writing to it until the lock is released.

3. **Read Lock and Write Lock:**

- **Read Lock:** Similar to a shared lock, allowing multiple transactions to read a resource concurrently.

- **Write Lock:** Similar to an exclusive lock, preventing any other transactions from reading or writing to a resource.

- **Example:** In a multi-user document editing system, multiple users may have read locks to view the document simultaneously. However, when one user is actively editing, a write lock is acquired to prevent others from modifying the document.



#### 4. **Read-Write Lock (or Multiple Read, Single Write Lock):**

- **Purpose:** Allows multiple transactions to acquire a read lock simultaneously, but only one transaction can acquire a write lock at a time.
- **Example:** In a caching system, multiple threads can read the cached data simultaneously using read locks. However, when updating or refreshing the cache, a write lock is acquired to ensure exclusive access during the update.

#### 5. **Deadlock:**

- **Purpose:** A situation where two or more transactions are unable to proceed because each is waiting for the other to release a lock.
- **Example:** Transaction A holds a lock that Transaction B needs, and vice versa. Both transactions are stuck, waiting for the other to release the required lock.

Understanding and managing locks is crucial in concurrent programming and database systems to ensure data consistency and prevent conflicts among multiple transactions.

#### 9. What is starvation? What are two conditions to avoid starvation?

In the context of a Database Management System (DBMS), "starvation" refers to a situation where a transaction or process is unable to proceed because it is consistently being denied access to a resource it needs, despite the resource being available. Starvation can occur due to resource allocation policies that prioritize certain transactions over others.

Two conditions to avoid starvation in a DBMS are:

1. **Fairness:** Ensure that all transactions or processes have a fair chance to access the required resources. This means preventing a single transaction from monopolizing resources indefinitely.
2. **Priority Scheduling:** Implement a scheduling or resource allocation policy that considers the priority of transactions. This helps in preventing low-priority transactions from being continuously blocked by high-priority ones, ensuring a more equitable distribution of resources.

#### 10. Explain strategies for deadlock prevention.

Deadlock prevention in Database Management Systems (DBMS) involves implementing strategies to ensure that deadlocks, which occur when transactions are blocked indefinitely, do not occur. Here are some key strategies for deadlock prevention:

##### 1. **Lock Hierarchy:**

- Establish a hierarchy for acquiring locks and ensure that transactions always request locks in the same order. This helps prevent circular waiting, a common cause of deadlocks.

2. **\*\*Timeouts:\*\***

- Set timeouts for transactions so that if a transaction cannot obtain all required locks within a specified time, it releases the locks it holds and restarts. This helps break potential deadlock situations.

3. **\*\*Wait-Die and Wound-Wait Schemes:\*\***

- Implement concurrency control schemes like Wait-Die and Wound-Wait to manage the interaction between transactions with conflicting lock requests. These schemes dynamically adjust the waiting or aborting of transactions based on their timestamps.

4. **\*\*Two-Phase Locking (2PL):\*\***

- Enforce the two-phase locking protocol, where transactions acquire locks in two phases (a growing phase and a shrinking phase). This protocol helps ensure that a transaction does not release any locks once it has acquired a lock.

5. **\*\*Resource Allocation Graph (RAG):\*\***

- Use a resource allocation graph to represent the relationships between transactions and the resources they request. Regularly check the graph for cycles, and if a cycle is detected, take corrective actions such as aborting one of the transactions involved.

6. **\*\*Dynamic Lock Allocation:\*\***

- Dynamically allocate locks to transactions based on their runtime behavior. This can involve releasing locks early or requesting additional locks based on the actual needs of the transactions during execution.

7. **\*\*Transaction Scheduling:\*\***

- Schedule transactions in a way that minimizes the likelihood of deadlocks. Smart scheduling algorithms can help in reducing the chances of circular waiting.

8. **\*\*Avoidance of Circular Waits:\*\***

- Ensure that transactions do not form circular wait chains. This can be achieved by assigning unique priorities to transactions or by preventing transactions with lower priorities from waiting for resources held by transactions with higher priorities.

By implementing these strategies, DBMS can significantly reduce the occurrence of deadlocks and improve the overall efficiency and reliability of the database system.

11. Explain Lock modes for multiple granularity with compatibility matrix.

In the context of databases and concurrency control, lock modes for multiple granularity refer to the different levels at which locks can be applied to data. The compatibility matrix defines which lock modes are compatible with each other to avoid conflicts and ensure data consistency. Here's a brief explanation:

**\*\*Lock Modes:\*\***

1. **\*\*Coarse-Grained Locks:\*\*** Lock entire data structures (e.g., entire tables).
2. **\*\*Medium-Grained Locks:\*\*** Lock a subset of a data structure (e.g., a range of rows in a table).
3. **\*\*Fine-Grained Locks:\*\*** Lock individual data items (e.g., a specific row or column in a table).

**\*\*Compatibility Matrix:\*\***

- **\*\*Compatible:\*\*** Locks can coexist without issues.
- **\*\*Incompatible:\*\*** Locks conflict and cannot coexist.

**\*\*Example Compatibility Matrix:\*\***

```

...
+-----+-----+-----+
|          | Coarse-Grained | Medium-Grained |
|          | (Table-level)  | (Row/Range-level) |
+-----+-----+-----+
| Coarse-Grained | Compatible      | Incompatible    |
| Medium-Grained  | Incompatible    | Compatible      |
| Fine-Grained    | Incompatible    | Incompatible    |
+-----+-----+-----+
...

```

This matrix illustrates the compatibility between different lock modes. For example, if one transaction has a coarse-grained lock on a table, another transaction cannot acquire a medium or fine-grained lock on the same table simultaneously. However, two transactions with medium-grained locks on different ranges of the table can coexist.

The goal is to allow concurrent access to data while preventing conflicts that could lead to data inconsistencies or other issues in a multi-user environment. The compatibility matrix helps in designing a locking strategy that balances concurrency and data integrity.

12. Explain Thomas' Write protocol with example.

If you are referring to Thomas' Write Rule, which is a part of the ACID properties in database systems, I can provide a brief explanation:

Thomas' Write Rule ensures that a transaction that has written a particular data item is the only transaction that can write that item until it is committed or rolled back. In other words, if a transaction T1 has modified a data item, no other transaction can modify that same item until T1 has completed.

Here's a simplified example to illustrate Thomas' Write Rule:

1. Transaction T1 begins and modifies data item X.
2. While T1 is still in progress, no other transaction (e.g., T2) is allowed to modify X.
3. T1 either commits or rolls back its changes to X.
4. Only after T1 has completed, another transaction (e.g., T2) can modify X.

This rule helps maintain consistency and isolation in a database system by preventing concurrent modifications to the same data item. It's important for ensuring that the final state of the database reflects the correct outcome of committed transactions.