END TERM PAPER 2024 SOLUTION DBMD

Question 1 (Compulsory)
(Attempt all parts - 3x5 = 15 Marks)

a) Describe the components of database systems. (3 Marks)

Answer:

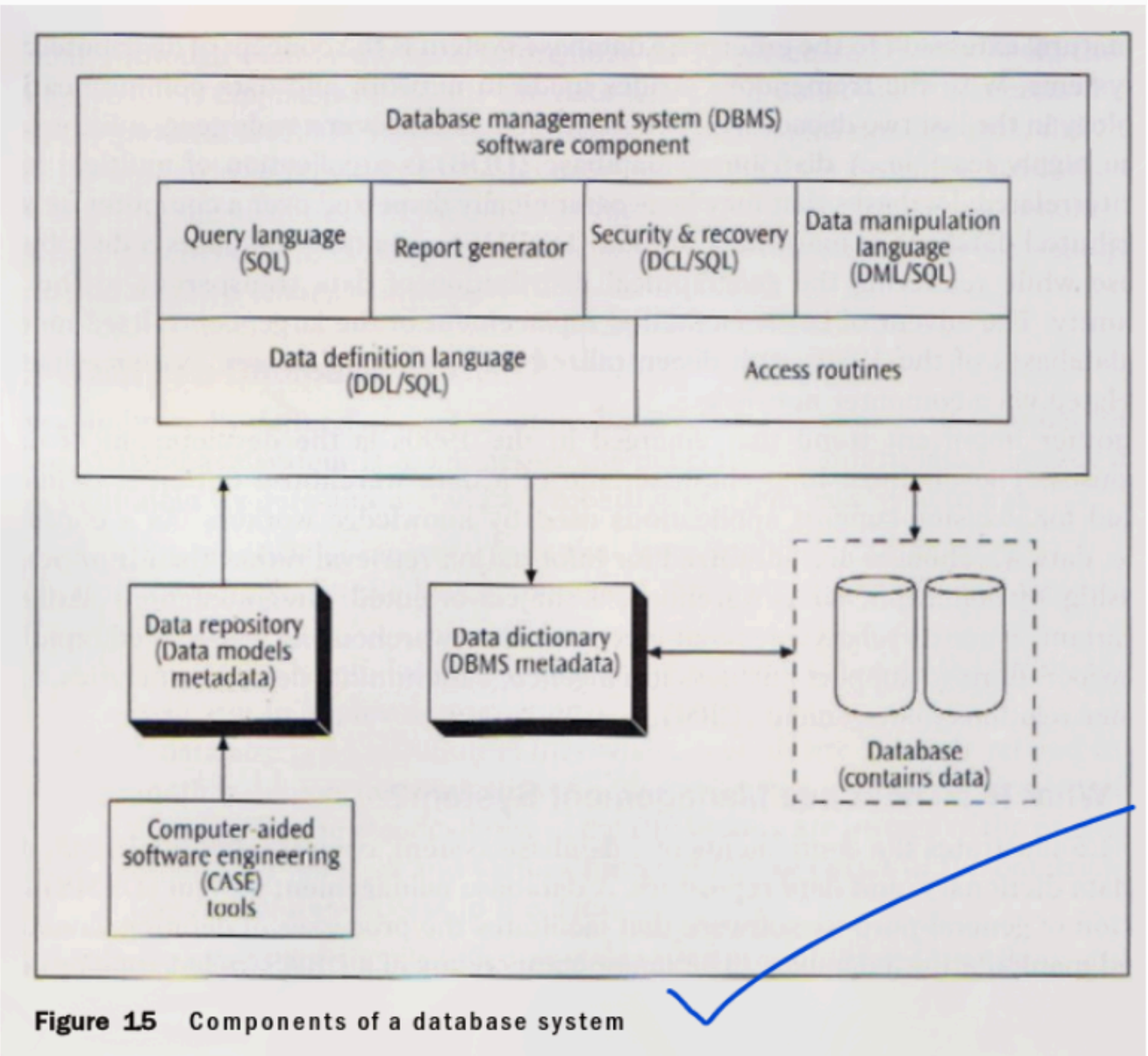


Figure 1.5 Components of a database system

A database system comprises several integrated components that work together to manage data efficiently. The main components are:

1. **Hardware:** Physical devices like servers, storage disks, network devices, etc., on which the database system runs.
2. **Software:**

- **Database Management System (DBMS):** The core software that defines, creates, manipulates, controls, and manages the database. It includes:
  - **Query Processor:** Interprets queries, creates an execution plan, and executes it. This includes DDL interpreter, DML compiler, and query evaluation engine.
  - **Storage Manager:** Manages physical storage, including buffer management, file management, and transaction management (concurrency control, recovery).
- **Application Programs & Utilities:** Software used to access and manipulate data, generate reports, perform backups, etc. Examples include report generators, data import/export tools.

3. **Data:** The actual information stored in the database, including operational data and metadata (data about data, stored in a data dictionary or repository).
4. **Users:**
   - **Database Administrators (DBAs):** Responsible for managing the overall database system, including security, backup, recovery, and performance tuning.
   - **Database Designers:** Responsible for defining the structure (schema) of the database.
   - **Application Programmers:** Develop applications that interact with the database.
   - **End Users:** Access the database to query, update data, and generate reports.
5. **Procedures:** Instructions and rules that govern the design and use of the database system, including login procedures, backup/recovery methods, etc.

(Based on: DBMD UNIT - 1 and 2 Notes, p.3, Figure 1.5 and related text; general DBMS knowledge)

---

b) Distinguish between primary key, candidate key and super key. (3 Marks)

Answer:

Feature	Superkey	Candidate Key	Primary Key
Definition	Any attribute or set of attributes that uniquely identifies a tuple (row) within a relation.	A minimal superkey. It is a superkey from which no attribute can be removed without losing its uniqueness property.	The candidate key chosen by the database designer to uniquely identify tuples in a relation.
Minimality	Not necessarily minimal. It may contain redundant attributes.	Must be minimal.	Must be minimal (as it's a chosen candidate key).
Uniqueness	Guarantees uniqueness for each tuple.	Guarantees uniqueness for each tuple.	Guarantees uniqueness for each tuple.
Number	A relation can have many superkeys.	A relation can have one or more candidate keys.	A relation has only one primary key.

Feature	Superkey	Candidate Key	Primary Key
Null Values	Generally, attributes part of a superkey should not be null, though not strictly enforced by definition alone (PK enforces no nulls).	Attributes forming a candidate key should ideally not accept null values.	Cannot contain null values (Entity Integrity Constraint).
Example	If {StudentID, CourseID} is a candidate key, then {StudentID, CourseID, StudentName} is a superkey.	In an Employee table, {EmployeeID} and {SocialSecurityNumber} could both be candidate keys.	From the candidate keys {EmployeeID} and {SocialSecurityNumber}, the DBA might choose {EmployeeID} as the primary key.

(Based on: DBMD UNIT - 1 and 2 Notes, p.8 "Unique Identifiers (Keys)", p.42 "Key Constraints")

---

c) Explain the advanced data manipulation using SQL. (3 Marks)

Answer:
Advanced Data Manipulation Language (DML) in SQL goes beyond simple `INSERT`, `SELECT`, `UPDATE`, `DELETE` on single rows. It often involves:

1. **Complex Queries:**
   - **Joins:** Combining data from multiple tables (e.g., `INNER JOIN`, `LEFT JOIN`, `RIGHT JOIN`, `FULL JOIN`, `CROSS JOIN`) based on related columns.
   - **Subqueries (Nested Queries):** Queries embedded within other SQL queries (in `WHERE`, `SELECT`, `FROM`, `HAVING` clauses) to perform complex filtering or calculations. This includes correlated subqueries.
   - **Aggregate Functions with Grouping:** Using functions like `COUNT()`, `SUM()`, `AVG()`, `MAX()`, `MIN()` with the `GROUP BY` clause to perform calculations on groups of rows, and `HAVING` to filter groups.
2. **Set Operations:** Combining results of two or more `SELECT` statements using `UNION`, `UNION ALL`, `INTERSECT`, `EXCEPT` (or `MINUS`).
3. **Window Functions (Advanced):** Performing calculations across a set of table rows that are somehow related to the current row. These are used for ranking, moving averages, cumulative sums, etc., without collapsing rows like `GROUP BY`.
4. **Common Table Expressions (CTEs):** Using the `WITH` clause to define temporary, named result sets that can be referenced within a single SQL statement, improving readability and modularity of complex queries.

5. **Bulk Operations:** Efficiently inserting, updating, or deleting large volumes of data, often using features specific to the DBMS or techniques like `INSERT INTO ... SELECT ...`.

These advanced features allow for sophisticated data retrieval, analysis, and modification, crucial for reporting, business intelligence, and complex application logic.

(Based on: DBMD UNIT - 3 and 4 Notes, p.12, p.15; general SQL knowledge extending beyond basic DML definitions in Unit 1 notes)

---

d) Explain the Data Control Language (DCL) commands. (3 Marks)

Answer:
Data Control Language (DCL) commands in SQL are used to manage permissions and control access to database objects. The primary DCL commands are:

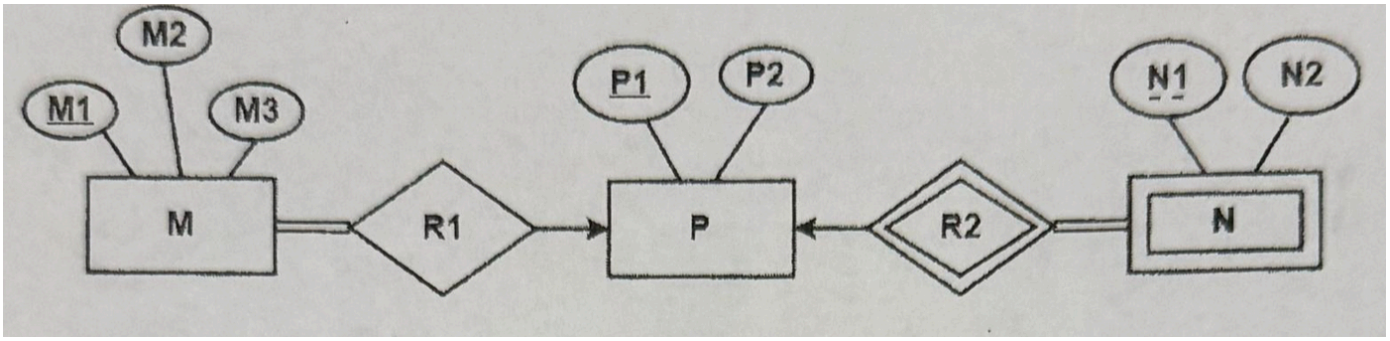
1. **GRANT:**
   - **Purpose:** Used to give specific privileges (access rights) to users or roles on database objects.
   - **Syntax (Simplified):**
     `GRANT privilege_list ON object_name TO user_list [WITH GRANT OPTION];`
   - **`privilege_list`**: Can include `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `REFERENCES`, `USAGE`, etc.
   - **`object_name`**: The database object like a table, view, or stored procedure.
   - **`user_list`**: The user(s) or role(s) receiving the privilege.
   - **`WITH GRANT OPTION`**: Allows the grantee to further grant the received privileges to other users.
   - **Example:** `GRANT SELECT, INSERT ON Employees TO 'john_doe';`
2. **REVOKE:**
   - **Purpose:** Used to remove previously granted or denied privileges from users or roles.
   - **Syntax (Simplified):**
     `REVOKE [GRANT OPTION FOR] privilege_list ON object_name FROM user_list [CASCADE | RESTRICT];`
   - **`GRANT OPTION FOR`**: Revokes only the ability to grant the privilege, not the privilege itself.
   - **`CASCADE`**: Revokes the privilege from the user and also from any other users to whom this user granted the privilege.
   - **`RESTRICT`**: (Often default) Fails if the privilege was passed on by the user.
   - **Example:** `REVOKE INSERT ON Employees FROM 'john_doe';`

DCL commands are fundamental for database security, ensuring that users can only perform authorized actions on data.

(Based on: DBMD UNIT - 1 and 2 Notes, p.4; DBMD UNIT - 3 and 4 Notes, p.32-34)

---

e) Find the minimum number of tables required to represent the given ER diagram in the relational model. (3 Marks)



Answer:

Okay, let's break down the entities and relationships to determine the minimum number of tables required:

1. Entity M (Strong Entity):

- Attributes: M1 (PK), M2, M3.
- **Rule:** Every strong entity gets its own table.
- **Table 1: Table_M (M1, M2, M3)**

2. Entity P (Strong Entity):

- Attributes: P1 (PK), P2.
- **Rule:** Every strong entity gets its own table.
- **Table 2: Table_P (P1, P2)**

3. Entity N (Weak Entity):

- Attributes: N1 (Partial Key), N2.
- **Rule:** A weak entity always gets its own table. The primary key of this table is a composite key formed by the primary key of its owner (strong) entity (P in this case, via R2) and its own partial key (N1).
- **Table 3: Table_N (P1_FK, N1, N2)**
  - Here, P1 from Table_P acts as a foreign key and is part of the primary key of Table_N.

Now let's consider the relationships:

4. Relationship R1 (between M and P) (1:N):

- This means one M instance can be related to many P instances, and one P instance is related to at most one M instance.
- **Rule for 1:N:** The primary key of the "one" side (M, which is M1) is added as a foreign key to the table of the "many" side (P).
- **Modification:** Table_P will now include M1 as a foreign key.

    - Table_P becomes: **Table_P (P1, P2, M1_FK)**
- No new table is created for a 1:N relationship.

5. Relationship R2 (identifying relationship) (between P and N) (M:1):

- This is an identifying relationship because N is a weak entity, and P is its owner. The (M:1) cardinality means many N instances relate to one P instance.
- **Rule for Identifying Relationship:** The mapping of the weak entity (N) already incorporates this relationship. The primary key of the owner entity (P1 from P) is included as part of the primary key of the weak entity's table (Table_N).
- This has already been handled when creating Table_N: **Table_N (P1_FK, N1, N2)**.
- No *additional* table is needed specifically for R2 because it's an integral part of defining the weak entity's table.

Consolidated List of Tables:

1. **Table_M (M1, M2, M3)**
2. **Table_P (P1, P2, M1_FK)** (M1_FK references Table_M)
3. **Table_N (P1_FK, N1, N2)** (P1_FK references Table_P, and {P1_FK, N1} forms the composite PK)

Therefore, the minimum number of tables required is **3**.

(Based on: ER to Relational Mapping rules covered in DBMD UNIT - 1 and 2 Notes, p.43-44)

UNIT-I
(Select one question from Q2 or Q3)

Question 2:

a) Explain the database systems architecture with a suitable diagram. (8 Marks)

Answer:

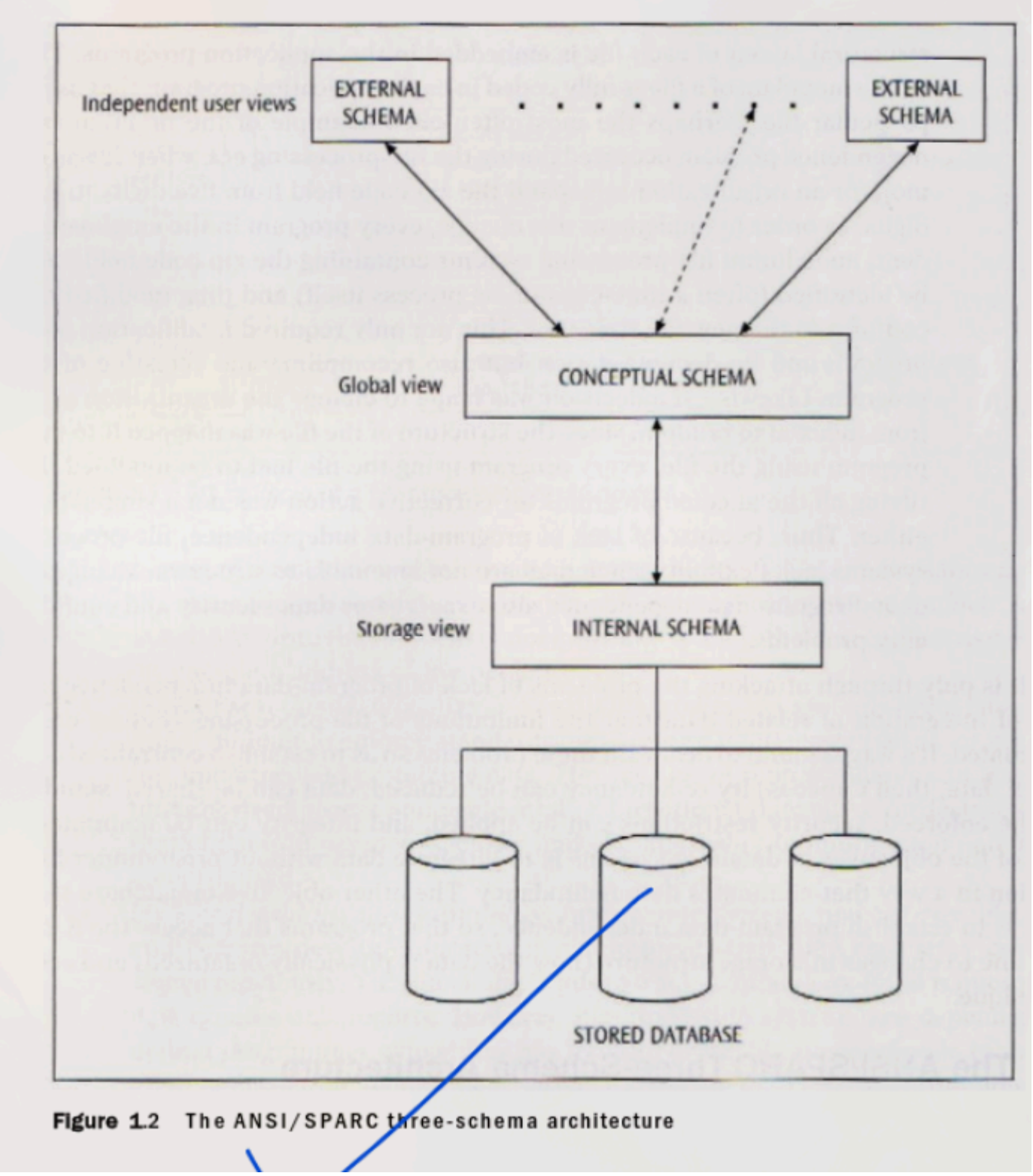


Figure 1.2 The ANSI/SPARC three-schema architecture

The most widely accepted architecture for database systems is the ANSI/SPARC three-schema architecture. Its primary goal is to achieve data independence, separating user views from the physical storage details. This architecture defines three levels:

1. **External Level (User Views / Subschemas):**
    - This is the highest level, closest to the users.
    - It describes the part of the database that is relevant to a particular user or group of users.
    - Different users can have different views of the database. For example, a user in HR might see employee salary information, while a user in project management might see employee skills and

project assignments, but not salaries.
    - Each view is defined by an external schema, which typically describes a subset of the database or data derived from it.
    - This level provides a level of security by hiding irrelevant or sensitive data from certain users.
2. **Conceptual Level (Global View / Conceptual Schema):**
    - This level provides a community view of the entire database.
    - It describes the logical structure of the whole database for all users. It defines all entities, attributes, relationships, constraints, and semantic information about the data.
    - The conceptual schema is technology-independent; it hides the details of physical storage structures and concentrates on describing what data is stored and how it is related.
    - This is the level at which database designers work. It provides a stable description of the data that doesn't change frequently.
3. **Internal Level (Physical View / Internal Schema):**
    - This is the lowest level, closest to the physical storage.
    - It describes how the data is physically stored on storage devices.
    - The internal schema specifies data structures, file organizations (e.g., B+-trees, hashing), access paths (e.g., indexes), data compression, and encryption techniques.
    - It is technology-dependent and deals with the efficiency of data access and storage.

Data Independence:
This architecture supports two types of data independence:

- **Logical Data Independence:** The ability to modify the conceptual schema without having to change the external schemas or application programs. For example, adding a new attribute or entity to the database.
- **Physical Data Independence:** The ability to modify the internal schema without having to change the conceptual or external schemas. For example, changing file organization or adding an index.

(Based on: DBMD UNIT - 1 and 2 Notes, p.2, Figure 1.2 and related text)

---

b) Describe the database design life cycle. (7 Marks)

Answer:

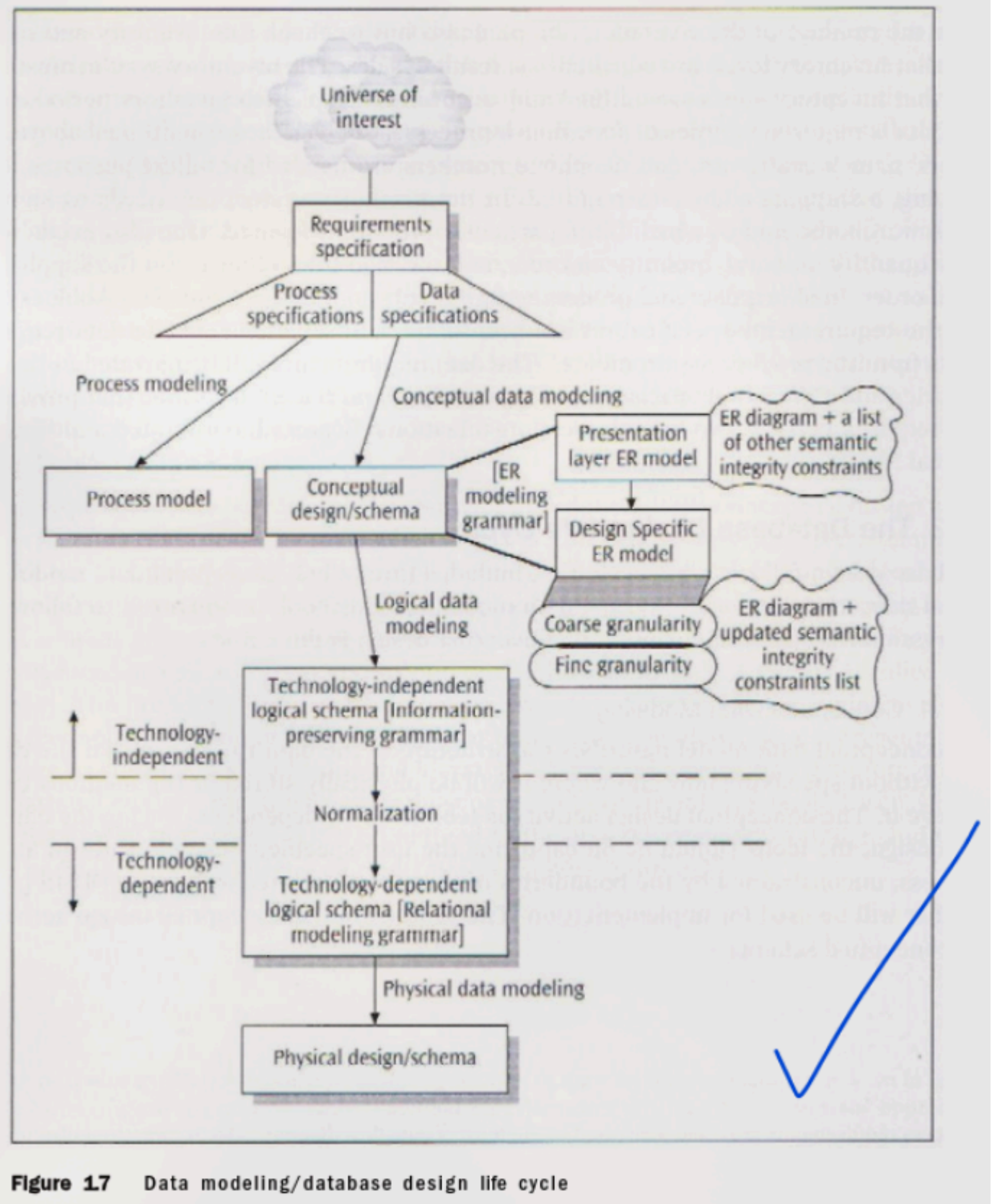


Figure 1.7 Data modeling/database design life cycle

The database design life cycle (DDLC) is a systematic process for designing, implementing, and maintaining a database system. It typically involves the following phases:

1. **Requirements Specification (and Planning):**
    - **Objective:** To understand and document the data requirements of the users and the intended application.
    - **Activities:**

        - Interviewing users, stakeholders, and domain experts.
        - Reviewing existing documents, forms, and reports.
        - Identifying objectives, data entities, processes, and business rules.
        - Defining the scope and boundaries of the database system.
    - **Output:** A detailed requirements specification document.
2. **Conceptual Data Modeling (Technology-Independent):**
    - **Objective:** To create a high-level description of the data structure, focusing on entities, attributes, and relationships, without considering physical storage details or specific DBMS.
    - **Activities:**
        - Translating requirements into a conceptual model (e.g., Entity-Relationship Diagram - ERD, or Enhanced ERD - EERD).
        - Identifying entity types, relationship types, attributes, and constraints (like cardinality, participation).
        - This phase often includes a presentation layer model for user communication and a design-specific layer for database design.
    - **Output:** Conceptual schema (e.g., an ER/EER diagram and supporting documentation). Validation with users is crucial.
3. **Logical Data Modeling (Technology-Dependent):**
    - **Objective:** To transform the conceptual schema into a logical model compatible with a chosen data model type (e.g., relational, network, hierarchical). For modern systems, this is typically the relational model.
    - **Activities:**
        - Mapping ER/EER constructs to relational tables (relations), attributes to columns, and defining primary and foreign keys.
        - Applying normalization techniques (e.g., 1NF, 2NF, 3NF, BCNF) to reduce data redundancy and improve data integrity.
    - **Output:** Logical schema (e.g., a set of normalized relational tables with their columns, primary keys, and foreign keys).
4. **Physical Data Modeling (DBMS-Specific):**
    - **Objective:** To specify the internal storage structures, access paths, and file organizations for the database, optimized for the chosen DBMS.
    - **Activities:**
        - Defining specific data types for columns (e.g., `VARCHAR`, `INT`, `DATE` in SQL).
        - Designing indexes (e.g., B+-tree, hash indexes) to improve query performance.
        - Considering file organization, data clustering, and partitioning strategies.
        - Estimating storage space and planning for security and recovery.

- **Output:** Physical schema (internal schema description, DDL statements).

5. **Implementation and Testing:**
    - **Objective:** To create the database based on the physical design and test its functionality.
    - **Activities:**
        - Writing DDL statements to create tables, indexes, views, etc.
        - Populating the database with initial data.
        - Developing and testing application programs that interact with the database.
        - Performing various tests (unit, integration, performance).
6. **Deployment, Operation, and Maintenance:**
    - **Objective:** To deploy the database system for operational use and maintain it over time.
    - **Activities:**
        - Deploying the database and applications.
        - Monitoring performance, tuning the database as needed.
        - Performing regular backups and planning for recovery.
        - Managing security and user access.
        - Making modifications or enhancements as requirements evolve.

(Based on: DBMD UNIT - 1 and 2 Notes, p.5, Figure 1.7 and related text)

Question 3:

a) Explain the data models available for the database modelling system. (8 Marks)

Answer:
Data models are conceptual tools used to describe the structure of a database, including data types, relationships, and constraints. Several data models have evolved over time:

1. **Hierarchical Data Model:**
    - **Structure:** Organizes data in a tree-like structure, where records are linked in parent-child relationships. Each child record can have only one parent, but a parent can have multiple children.
    - **Representation:** Data is represented as a collection of records connected by links.
    - **Example:** An organization chart where a manager (parent) has several employees (children) reporting to them.
    - **Advantages:** Simple to understand for some types of data, efficient for accessing data along predefined hierarchical paths, data integrity is maintained through parent-child links.
    - **Disadvantages:** Inflexible, as representing many-to-many relationships is complex and often requires data duplication. Data access is restricted to navigating the tree structure. Deleting a

parent automatically deletes its children.

- **Relevance:** Primarily of historical interest (e.g., IBM's IMS).

2. **Network Data Model:**
   - **Structure:** An extension of the hierarchical model, allowing records to have multiple parent records as well as multiple child records. Data is organized as a graph (network).
   - **Representation:** Records are called "nodes," and relationships are called "sets" (directed links).
   - **Example:** A student can enroll in multiple courses, and a course can have multiple students (a many-to-many relationship directly representable).
   - **Advantages:** More flexible than the hierarchical model, can represent complex relationships (including M:N) more naturally, reduces data redundancy compared to hierarchical.
   - **Disadvantages:** Still complex to design and manage, data access requires navigating through record pointers, application programs are complex and tied to the database structure.
   - **Relevance:** Also largely historical (e.g., IDMS).
3. **Relational Data Model:**
   - **Structure:** Represents data as a collection of tables (relations). Each table has rows (tuples) and columns (attributes).
   - **Representation:** Data is stored in two-dimensional tables. Relationships between tables are established through common columns (foreign keys referencing primary keys).
   - **Example:** An `Employees` table and a `Departments` table, where `Employees` has a `DepartmentID` foreign key referencing the `DepartmentID` primary key in `Departments`.
   - **Advantages:** Simple and intuitive, provides a high degree of data independence (logical and physical), flexible querying using SQL, strong mathematical foundation (relational algebra and calculus), enforces data integrity through constraints.
   - **Disadvantages:** Can sometimes lead to slower performance for very complex queries involving many joins if not properly optimized.
   - **Relevance:** The most widely used data model today for transactional systems (e.g., Oracle, MySQL, SQL Server, PostgreSQL).
4. **Entity-Relationship (ER) Data Model (Conceptual Model):**
   - **Structure:** A high-level conceptual data model used for database design. It views the real world as a collection of basic objects (entities) and relationships among these objects.
   - **Representation:** Entities are represented by rectangles, attributes by ovals, and relationships by diamonds. Cardinality and participation constraints define the nature of relationships.
   - **Advantages:** Easy to understand and communicate database design to non-technical users, provides a clear graphical representation of the database structure.
   - **Disadvantages:** Not directly implemented by DBMS; it's a design tool that is then mapped to a logical model like relational.
   - **Relevance:** Standard tool for conceptual database design.

5. **Enhanced Entity-Relationship (EER) Data Model (Conceptual Model):**
    - **Structure:** An extension of the ER model that includes concepts like superclasses/subclasses, specialization/generalization, inheritance, and categorization (union types).
    - **Advantages:** Allows for more precise and semantic modeling of complex data requirements.
    - **Relevance:** Used for designing more complex databases with inheritance hierarchies.
6. **Object-Oriented Data Model:**
    - **Structure:** Data is stored as objects, which are instances of classes. Objects encapsulate both data (attributes) and behavior (methods/operations). Supports concepts like inheritance, polymorphism.
    - **Advantages:** Can directly represent complex data types and relationships found in object-oriented programming, good for applications like CAD/CAM, multimedia.
    - **Disadvantages:** Less mature than the relational model, querying can be more complex, less widespread adoption for general-purpose databases.
    - **Relevance:** Used in Object-Oriented DBMS (OODBMS) and Object-Relational DBMS (ORDBMS).
7. **Object-Relational Data Model:**
    - **Structure:** A hybrid model that combines features of the relational model with object-oriented concepts. Allows user-defined data types, complex objects, inheritance within a relational framework.
    - **Advantages:** Extends the power of the relational model while maintaining compatibility with SQL.
    - **Relevance:** Many modern relational DBMSs (e.g., PostgreSQL, Oracle) incorporate object-Relational features.
8. **NoSQL Data Models (e.g., Document, Key-Value, Column-family, Graph):**
    - **Structure:** A broad category of database models that are non-relational. They are designed for scalability, flexibility, and high performance for specific types of data and workloads (e.g., big data, real-time web applications).
        - **Document:** Stores data in documents (e.g., JSON, BSON, XML).
        - **Key-Value:** Simple model storing data as (key, value) pairs.
        - **Column-family:** Stores data in columns rather than rows, good for sparse data.
        - **Graph:** Uses nodes, edges, and properties to represent and store data, ideal for highly connected data.
    - **Advantages:** High scalability and availability, flexible schema, good for unstructured or semi-structured data.
    - **Disadvantages:** Often provide eventual consistency rather than ACID transactions, querying capabilities can be limited compared to SQL.

- **Relevance:** Increasingly popular for specific use cases where traditional RDBMSs face limitations.

(Based on: General knowledge of data models; ER and EER are covered in DBMD UNIT - 1 and 2 Notes, p.6, p.27. Relational model is the basis for much of the notes.)

---

b) Illustrate the design issues in ER & EER modelling. (7 Marks)

Answer:
Designing an effective ER (Entity-Relationship) or EER (Enhanced Entity-Relationship) model requires careful consideration of several design issues to accurately represent the real-world domain and create a robust database schema. Common design issues include:

1. **Choosing Entity vs. Attribute:**
   - **Issue:** Deciding whether a real-world concept should be modeled as an entity type or an attribute of an existing entity type.
   - **Guideline:** If the concept has its own descriptive properties (attributes) or participates in relationships independently, it's better modeled as an entity. If it's a simple property describing another entity and doesn't have attributes of its own or participate in relationships, it's an attribute.
   - **Example:** Should "Address" be an attribute of `Employee` or a separate `Address` entity? If an employee can have multiple addresses, or if addresses need to store complex details (like geocodes, validity dates) and are shared by multiple entities, then `Address` as an entity is better. If it's just a single, simple street address for an employee, an attribute might suffice.
2. **Choosing Entity vs. Relationship:**
   - **Issue:** Deciding whether a concept is better represented as an entity or a relationship. This often arises with M:N relationships that have their own attributes.
   - **Guideline:** If an association between entities has descriptive attributes, it's often better to model the association itself as an (associative) entity.
   - **Example:** A `WORKS_ON` relationship between `Employee` and `Project` might have an attribute `HoursWorked`. This can be modeled as an M:N relationship with an attribute, or `WORKS_ON` can be promoted to an associative entity `Assignment` with attributes `HoursWorked`, and 1:N relationships to `Employee` and `Project`.
3. **Binary vs. Higher-Degree Relationships (Ternary, N-ary):**
   - **Issue:** Deciding the degree of a relationship (number of participating entity types). Most relationships are binary.
   - **Guideline:** Use higher-degree relationships sparingly and only when the relationship genuinely and irreducibly involves more than two entities simultaneously. Often, a ternary relationship can be decomposed into several binary relationships, possibly with an intermediate associative entity.

- **Example:** A relationship `SUPPLIES` involving `Supplier`, `Part`, and `Project`. If a supplier supplies a specific part *only* for a specific project, it's a true ternary relationship. If a supplier supplies parts, and projects use parts, it might be better modeled with binary relationships.
- *(See DBMD UNIT - 1 and 2 Notes, p.33-34 on ternary relationships).*

4. **Use of Weak Entities:**
   - **Issue:** Deciding when to use a weak entity type. A weak entity cannot be identified by its own attributes alone and relies on its relationship with an owner (strong) entity for identification.
   - **Guideline:** Use weak entities when an entity's existence depends on another entity, and its primary key includes the primary key of the owner entity.
   - **Example:** `Dependents` of an `Employee`. A dependent cannot exist without the employee, and might be identified by `(EmployeeID, DependentName)`.
   - *(See DBMD UNIT - 1 and 2 Notes, p.18).*
5. **When to Use EER Constructs (Specialization/Generalization, Categorization):**
   - **Issue:** Deciding whether to use superclass/subclass hierarchies or categories.
   - **Specialization/Generalization (Is-A):** Use when an entity type has distinct subgroups (subclasses) that have some common attributes/relationships (inherited from the superclass) and also some specific attributes/relationships.
     - **Example:** `Employee` as a superclass with subclasses `SalariedEmployee`, `HourlyEmployee`.
   - **Categorization (Union Type):** Use when a subclass represents a collection of entities that is a subset of the *union* of distinct entity types. The subclass (category) has a relationship with multiple, different superclasses.
     - **Example:** `VehicleOwner` could be a category whose members are either a `Person` or a `Company`.
   - *(See DBMD UNIT - 1 and 2 Notes, p.27, p.29, p.32-33).*
   - **Constraints for Specialization:** Deciding on disjointness (d/o - disjoint/overlapping) and completeness (total/partial participation of superclass in subclasses).
6. **Handling M:N Relationships and Multi-valued Attributes in Design-Specific ER:**
   - **Issue:** While conceptual ER allows direct representation, for mapping to relational models, M:N relationships and multi-valued attributes need decomposition.
   - **Guideline (Design-Specific ER):**
     - M:N relationships are decomposed into two 1:N relationships with an intermediate associative entity.
     - Multi-valued attributes are typically transformed into a separate entity related to the original entity with a 1:N relationship.
   - *(See DBMD UNIT - 1 and 2 Notes, p.24, p.26 for M:N and multi-valued attribute resolution).*
7. **Validation of Conceptual Design (Connection Traps):**

- **Issue:** The structure of the ER diagram can sometimes lead to misinterpretation or ambiguity about how entities are related.
- **Fan Trap:** Occurs when a model represents a relationship between entity types, but the pathway between certain entity occurrences is ambiguous because two or more 1:N relationships fan out from the same entity.
  - **Example:** A `Division` has many `Staff` and many `Branches`. The ER diagram doesn't directly show which staff work at which branch. (The solution often involves restructuring or adding direct relationships).
- **Chasm Trap:** Occurs when a model suggests the existence of a relationship between entity types, but the pathway does not exist for certain entity occurrences, often due to optional participation in relationships along the path.
  - **Example:** `Branch` has `Staff`, and `Staff` manage `Property`. If participation of `Staff` in managing `Property` is optional, you can't find all properties at a branch if some staff manage no properties.
- *(See DBMD UNIT - 1 and 2 Notes, p.37-40 for Connection Traps).*

Addressing these design issues thoughtfully leads to a well-structured, accurate, and robust database model.

UNIT-II
(Select one question from Q4 or Q5)

Question 4:

a) Explain the ER model and EER Model to map with logical schema. (8 Marks)

Answer:
Mapping Entity-Relationship (ER) and Enhanced Entity-Relationship (EER) models to a logical schema (typically a relational schema) involves a set of rules to convert conceptual constructs into tables, columns, and keys.

Mapping ER Model Constructs to Relational Schema:

1. **Strong Entity Types:**
   - For each strong entity type, create a new relation (table).
   - The attributes of the entity become columns in the table.
   - Choose one of the candidate keys as the primary key for the table.
   - Simple attributes map directly. Composite attributes are mapped by including their simple component attributes. Stored attributes are included. Derived attributes are generally not stored but can be calculated.
2. **Weak Entity Types:**

- For each weak entity type, create a new relation.
- Include all attributes of the weak entity as columns.
- The primary key of this new relation is formed by the primary key of its owner (strong) entity (as a foreign key) plus the partial key (discriminator) of the weak entity.

3. **1:1 Relationship Types:**
   - **Option 1 (Foreign Key):** Choose one of the participating entity tables (e.g., if one side has total participation, make it that side) and add the primary key of the other entity table as a foreign key. This foreign key column should be constrained to be unique.
   - **Option 2 (Merged Relation):** If both sides have total participation, consider merging the two entity types into a single relation. This is less common.
   - **Option 3 (Relationship Relation):** If both participations are partial, or to avoid nulls, create a separate relation for the relationship. Its primary key would be the PK of one entity, and it would include the PK of the other as a (unique) FK. Attributes of the relationship go into this new table.
4. **1:N (One-to-Many) Relationship Types:**
   - **Foreign Key Approach (Standard):** Identify the entity type on the 'N' (many) side. In the relation corresponding to the N-side entity, include the primary key of the '1' (one) side entity as a foreign key.
   - Attributes of the 1:N relationship are also included in the N-side relation.
5. **M:N (Many-to-Many) Relationship Types:**
   - Create a new relation (junction/associative table) specifically for the M:N relationship.
   - The primary key of this new relation is a composite key formed by the primary keys of the two participating entity types (both acting as foreign keys).
   - Any attributes of the M:N relationship become columns in this new relation.
6. **Multi-valued Attributes:**
   - Create a new relation for the multi-valued attribute.
   - This new relation will include the primary key of the entity type to which the attribute belongs (as a foreign key) and the multi-valued attribute itself.
   - The primary key of this new relation is the combination of the foreign key and the multi-valued attribute.

Mapping EER Model Constructs to Relational Schema:

1. **Specialization/Generalization (Superclass/Subclass - SC/sc):**
   - **Option 1 (Multiple Relations - SC and sc's):** Create a relation for the superclass (SC) and a separate relation for each subclass (sc). The primary key of SC is also the primary key for each sc relation, and in sc relations, it also acts as a foreign key referencing SC. This is good for disjoint subclasses with specific attributes.

- **Option 2 (Single Relation - SC only):** Create one relation for the superclass. Include all attributes of the SC and all attributes of all its subclasses. Use nulls for attributes not applicable to a particular instance. Add a "type" attribute to distinguish between subclass instances. Good for overlapping subclasses or when subclasses have few specific attributes.
- **Option 3 (Multiple Relations - sc's only):** Create a relation for each subclass. Include all inherited attributes from the SC in each subclass relation. This option is viable only if the SC participation is total and disjoint. It can lead to redundancy of SC attributes.

2. **Specialization Hierarchy/Lattice (Multiple Inheritance):**
    - Apply the chosen SC/sc mapping option recursively. For a shared subclass (lattice), its relation will have multiple foreign keys if Option 1 is used, one for each superclass path it participates in.
3. **Categorization (Union Type):**
    - Create a relation for the category (subclass).
    - Create relations for each of the superclasses.
    - The primary key of the category is usually a surrogate key (a newly generated ID). This PK is then included as a foreign key in each superclass relation to link it to the category instance (this is different from specialization where subclasses inherit PK from superclass). This mapping is complex and often avoided if possible.

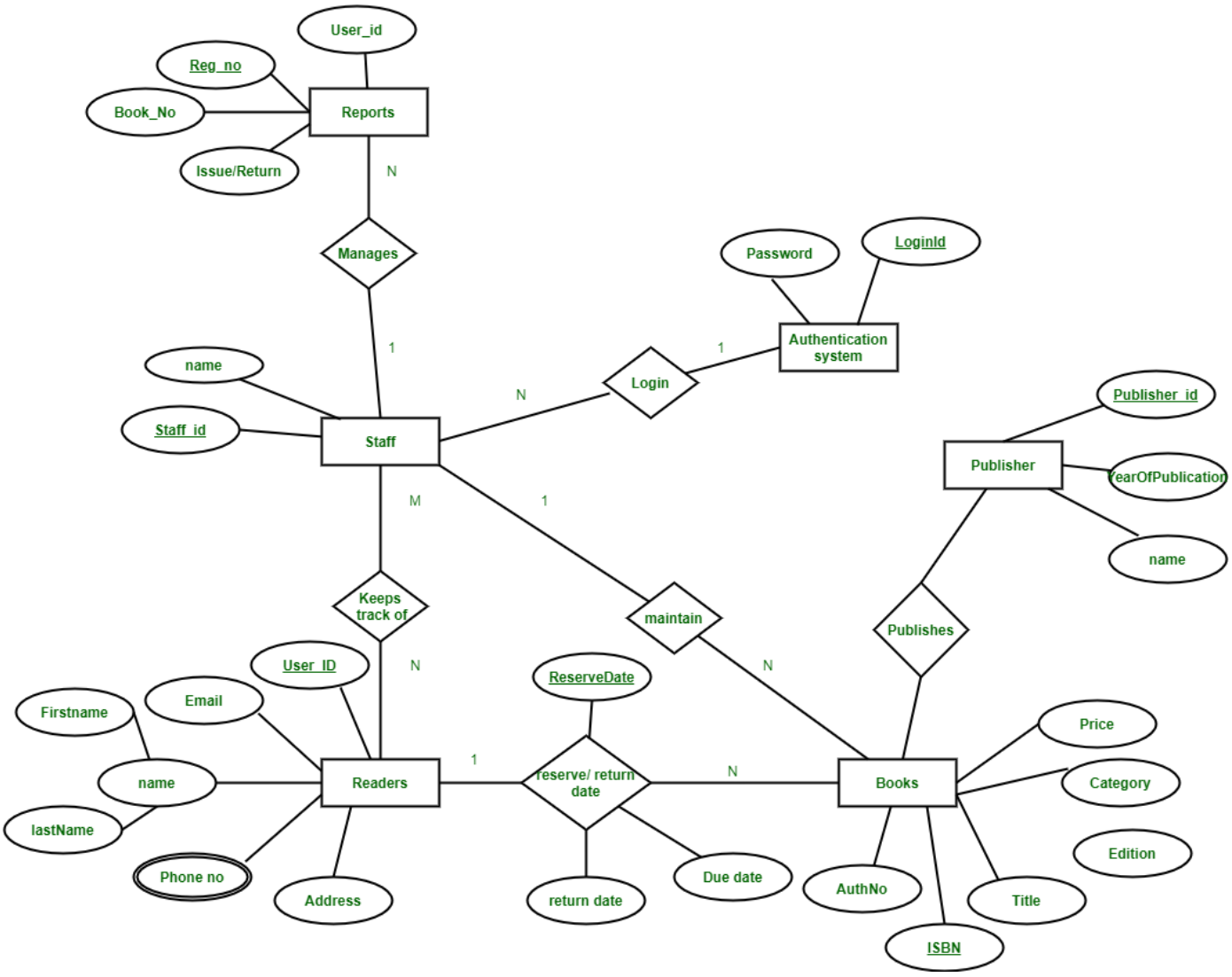
(Based on: DBMD UNIT - 1 and 2 Notes, p.43-44 for ER, p.45, p.47-49 for EER)

---

b) Construct an ER diagram for the Library Management System covering all major activities and map its logical schema. (7 Marks)

Answer:

ER Diagram for Library Management System:



1. **Primary Keys (PKs):** Attributes like `User_id` (for Reports), `Staff_id`, `LoginId`, `Publisher_id`, `User_ID` (for Readers), and `ISBN` (for Books) are assumed to be primary keys for their respective entities.
2. **"Reports" Entity:** This entity seems to represent transactions like issuing or returning books. It has attributes `User_id` (likely a reader), `Book_No` (likely a book), and `Issue/Return` (status). `Reg_no` could be a transaction ID.
3. **"reserve/return date" Relationship:** This M:N relationship between Readers and Books has attributes (`ReserveDate`, `return date`, `Due date`), so it will become a separate table.
4. **"maintain" Relationship:** This connects `Staff` to the "reserve/return date" relationship, implying a staff member is involved in the reservation/loan process.
5. **"Phone no" for Readers:** The double oval indicates a multi-valued attribute, requiring a separate table.
6. **"name" Attributes:**
   - For `Staff`, `name` will be split into `StaffFirstName` and `StaffLastName`.
   - For `Readers`, `FirstName` and `lastName` are already provided; `name` might be redundant or a full name concatenation, but we'll stick to `FirstName` and `LastName`.

7. **`AuthNo` in `Books`:** Without a separate `Author` entity linked via `AuthNo`, this will be treated as a simple attribute of `Books`. In a more complete system, this would likely be a foreign key to an `Authors` table.
8. **`YearOfPublication`:** As it's directly connected to `Publisher`, it's an attribute of `Publisher` (e.g., publisher's founding year or a general publication year they handle, which is less common than being an attribute of a specific book). If it were the publication year of a *book*, it should be an attribute of `Books` or the `Publishes` relationship.

Logical Schema (Relational Tables):

1. AuthenticationSystem_Table
* `LoginID` (PK, Data Type: VARCHAR(50) or as appropriate)
* `Password` (Data Type: VARCHAR(255) - should be hashed in a real system)

2. Staff_Table
* `StaffID` (PK, Data Type: VARCHAR(20) or INT)
* `StaffFirstName` (Data Type: VARCHAR(50), NOT NULL)
* `StaffLastName` (Data Type: VARCHAR(50), NOT NULL)
* `LoginID` (FK, Data Type: VARCHAR(50), References AuthenticationSystem_Table(LoginID))
* *Constraint: Could be UNIQUE if one staff has one login.*

3. Publisher_Table
* `PublisherID` (PK, Data Type: VARCHAR(20) or INT)
* `PublisherName` (Data Type: VARCHAR(100), NOT NULL, UNIQUE)
* `YearOfPublication` (Data Type: INT) *(Interpreted as an attribute of the publisher itself)*

4. Books_Table
* `ISBN` (PK, Data Type: VARCHAR(13))
* `Title` (Data Type: VARCHAR(255), NOT NULL)
* `AuthNo` (Data Type: VARCHAR(50)) *(Would ideally be an FK to an Authors table)*
* `Edition` (Data Type: VARCHAR(50))
* `Category` (Data Type: VARCHAR(50))
* `Price` (Data Type: DECIMAL(10,2))
* `PublisherID` (FK, Data Type: VARCHAR(20) or INT, References Publisher_Table(PublisherID))

5. Readers_Table
* `UserID` (PK, Data Type: VARCHAR(20) or INT)
* `FirstName` (Data Type: VARCHAR(50), NOT NULL)
* `LastName` (Data Type: VARCHAR(50), NOT NULL)
* `Email` (Data Type: VARCHAR(100), UNIQUE)
* `Address` (Data Type: VARCHAR(255))

6. Reader_Phones_Table (For multi-valued Phone no)
* `UserID` (FK, PK_part, Data Type: VARCHAR(20) or INT, References Readers_Table(UserID))
* `PhoneNumber` (PK_part, Data Type: VARCHAR(20))
* *Primary Key: (`UserID`, `PhoneNumber`)*

7. Book_Loan_Transaction_Table (Derived from "Reports" entity and "reserve/return date" relationship, incorporating "Manages" and "maintain")
* `TransactionID` (PK, Data Type: INT or VARCHAR(30), e.g., `Reg_no` from "Reports")
* `Reader_UserID` (FK, Data Type: VARCHAR(20) or INT, References Readers_Table(UserID), NOT NULL)
* `Book_ISBN` (FK, Data Type: VARCHAR(13), References Books_Table(ISBN), NOT NULL)
* `Issuing_StaffID` (FK, Data Type: VARCHAR(20) or INT, References Staff_Table(StaffID), NOT NULL)
* *(This staff member manages the issue/return and maintains the record)*
* `ReserveDate` (Data Type: DATE) *(Nullable, if not all loans start with a reservation)*
* `IssueDate` (Data Type: DATE) *(Derived from "Issue/Return" if it's an issue)*
* `DueDate` (Data Type: DATE, NOT NULL if issued)
* `ReturnDate` (Data Type: DATE) *(Nullable)*
* `Status` (Data Type: VARCHAR(20), e.g., 'Issued', 'Returned', 'Reserved', 'Overdue'. Captures `Issue/Return` attribute.)
* *Alternatively, instead of TransactionID, a composite PK like (`Reader_UserID`, `Book_ISBN`, `IssueDate`) could be used if a user can't borrow the same book on the same day. A surrogate `TransactionID` is often cleaner for transaction tables.*

8. Staff_Reader_Monitoring_Table (Derived from "Keeps track of" M:N relationship)
* `StaffID` (FK, PK_part, Data Type: VARCHAR(20) or INT, References Staff_Table(StaffID))
* `Reader_UserID` (FK, PK_part, Data Type: VARCHAR(20) or INT, References Readers_Table(UserID))
* `MonitoringStartDate` (Data Type: DATE) *(Optional attribute to give context to the tracking)*
* *Primary Key: (`StaffID`, `Reader_UserID`)*

(Based on ER design principles and mapping rules from DBMD UNIT - 1 and 2 Notes)

Question 5:

a) Define Normalization. Explain the types of Normalization. (8 Marks)

Answer:
Definition of Normalization:
Normalization is the process of organizing data in a database to reduce redundancy and improve data integrity. It involves decomposing larger tables into smaller, well-structured tables and defining relationships between them. The primary goals of normalization are:

- Minimizing data redundancy (avoiding storing the same piece of information multiple times).

- Eliminating undesirable characteristics like Insertion, Update, and Deletion Anomalies.
- Ensuring data dependencies make sense (i.e., attributes should only depend on the primary key).
- Producing a more flexible and maintainable database design.

Normalization is achieved by applying a series of rules or tests, leading to different "normal forms."

Types of Normalization (Normal Forms):

1. **First Normal Form (1NF):**
    - **Rule:** A relation is in 1NF if all its attribute values are atomic. This means each cell in the table must contain a single, indivisible value, and there should be no repeating groups or multi-valued attributes within a single row/column intersection.
    - **Essentially:** Eliminates repeating groups and multi-valued attributes by placing them in separate tables or by creating separate rows for each value.
    - *(See DBMD UNIT - 1 and 2 Notes, p.51-52).*
2. **Second Normal Form (2NF):**
    - **Rule:** A relation is in 2NF if it is in 1NF and every non-primary-key attribute is fully functionally dependent on the entire primary key. This means there are no partial dependencies (where a non-key attribute depends on only a part of a composite primary key).
    - **Essentially:** Removes partial dependencies by decomposing the table. If a non-key attribute depends on only part of a composite PK, that part of the PK and the dependent attribute(s) are moved to a new table.
    - *(See DBMD UNIT - 1 and 2 Notes, p.52-53).*
3. **Third Normal Form (3NF):**
    - **Rule:** A relation is in 3NF if it is in 2NF and there are no transitive dependencies. A transitive dependency exists when a non-primary-key attribute depends on another non-primary-key attribute, which in turn depends on the primary key.
    - **Essentially:** Removes transitive dependencies. If `A -> B` and `B -> C` (where A is PK, B and C are non-key), then `B` and `C` are moved to a new table where B becomes the PK.
    - *(See DBMD UNIT - 1 and 2 Notes, p.53-54).*
4. **Boyce-Codd Normal Form (BCNF):**
    - **Rule:** A relation is in BCNF if it is in 3NF and for every non-trivial functional dependency `X -> Y`, `X` must be a superkey (or candidate key) of the relation. BCNF is a stricter version of 3NF.
    - **Essentially:** Addresses certain anomalies that can still exist in 3NF tables if there are multiple overlapping candidate keys.
    - *(See DBMD UNIT - 1 and 2 Notes, p.54-55).*
5. **Fourth Normal Form (4NF):**

- **Rule:** A relation is in 4NF if it is in BCNF and has no non-trivial multi-valued dependencies (MVDs), unless the determinant of the MVD is a superkey. An MVD `X ->> Y` means that Y values are determined by X independently of other attributes Z in the relation.
- **Essentially:** Decomposes tables to separate independent multi-valued facts about an entity.
- *(See DBMD UNIT - 1 and 2 Notes, p.55-57).*

6. **Fifth Normal Form (5NF / Project-Join Normal Form - PJ/NF):**
    - **Rule:** A relation is in 5NF if it is in 4NF and has no join dependencies (JDs) that are not implied by candidate keys. A JD means a relation can be losslessly decomposed into projections and reconstructed by joining them.
    - **Essentially:** Deals with more complex cases of redundancy that are not covered by MVDs. Ensures that the table cannot be decomposed further without loss of information, except based on candidate keys.
    - *(See DBMD UNIT - 1 and 2 Notes, p.57-58).*

Generally, achieving 3NF or BCNF is considered sufficient for most practical database designs, as higher normal forms can sometimes lead to an excessive number of tables and complex joins, potentially impacting performance.

---

b) Explain the Mapping of higher degree relationships. (7 Marks)

Answer:
Higher-degree relationships (ternary, quaternary, or n-ary) involve three or more entity types participating simultaneously in a single relationship instance. Mapping these to a relational schema requires careful consideration to accurately represent the associations and avoid loss of information.

The General Approach: Associative Entity (Junction Table)

The standard method for mapping any higher-degree relationship (degree > 2) is to create a new relation (often called an associative entity or junction table) to represent the relationship itself.

Steps for Mapping an N-ary Relationship:

1. **Create a New Relation:** For the n-ary relationship type `R`, create a new relation (table) `S`.
2. **Include Foreign Keys:** For each of the `n` participating entity types `E1, E2, ..., En`, include the primary key attribute(s) of `Ei` as foreign key attributes in the new relation `S`. These foreign keys will reference the primary keys of their respective entity tables.
3. **Determine the Primary Key of the New Relation `S`:**
    - The primary key of `S` is typically the combination of all the foreign key attributes included from the participating entity types. That is, `{PK(E1), PK(E2), ..., PK(En)}`.
    - This composite primary key ensures that each combination of participating entity instances in the relationship is unique.

    - In some cases, if the cardinality constraints on one or more sides of the n-ary relationship are '1', the primary key of `S` might not need to include all foreign keys. However, it's usually safest to start with the combination of all foreign keys.
4. **Map Attributes of the Relationship:** If the n-ary relationship `R` has its own attributes, these attributes become columns in the new relation `S`.
5. **Cardinality Constraints (More Complex):** Representing exact cardinality constraints (like `(min,max)`) for n-ary relationships directly in the relational schema using only PK/FK constraints is challenging. These often need to be enforced through application logic or complex `CHECK` constraints/triggers if the DBMS supports them effectively for this purpose.

Example: Ternary Relationship `SCHEDULE`
(Referencing Fig 5.3 from DBMD UNIT - 1 and 2 Notes, p.34, and general mapping principles)

Let's assume a ternary relationship `SCHEDULE` involves three entities: `INSTRUCTOR`, `COURSE`, and `QUARTER`. An instructor teaches a specific course in a specific quarter.

- `INSTRUCTOR (InstructorID_PK, Name, ...)`
- `COURSE (CourseID_PK, CourseName, Credits, ...)`
- `QUARTER (QuarterID_PK, QuarterName, Year, ...)`

The `SCHEDULE` relationship might have an attribute like `RoomNo`.

Mapping `SCHEDULE` to a Relational Table:

A new table, say `SCHEDULE_TABLE`, is created:

```
SCHEDULE_TABLE (
    InstructorID_FK INT,       -- Foreign key referencing
INSTRUCTOR(InstructorID_PK)
    CourseID_FK VARCHAR(10),   -- Foreign key referencing COURSE(CourseID_PK)
    QuarterID_FK VARCHAR(10),  -- Foreign key referencing QUARTER(QuarterID_PK)
    RoomNo VARCHAR(10),        -- Attribute of the SCHEDULE relationship
    PRIMARY KEY (InstructorID_FK, CourseID_FK, QuarterID_FK),
    FOREIGN KEY (InstructorID_FK) REFERENCES INSTRUCTOR(InstructorID_PK),
    FOREIGN KEY (CourseID_FK) REFERENCES COURSE(CourseID_PK),
    FOREIGN KEY (QuarterID_FK) REFERENCES QUARTER(QuarterID_PK)
);
```

Considerations for Higher-Degree Relationships:

- **Decomposition:** Before mapping, always evaluate if a higher-degree relationship can be meaningfully decomposed into several binary relationships. This is often preferred if it doesn't lose semantics. For instance, if an instructor is assigned to teach a course (binary), and that course offering is scheduled in a quarter (binary), this might be an alternative if the three are not intrinsically

linked simultaneously in all cases. However, if the rule is "Instructor X teaches Course Y *specifically in* Quarter Z", then a ternary relationship is appropriate.

- **Conceptual Model First:** The notes suggest (DBMD UNIT - 1 and 2 Notes, p.50, "Mapping of Higher Degree Relationships") to "Decompose into a gerund (associative) entity type in the conceptual model first... Then map this gerund entity and its binary relationships to relational schema." This means conceptually turning the diamond (relationship) into a rectangle (associative entity) and then mapping that entity. The result in the relational model is the same as described above.
- **Aggregation:** Aggregation is a form of abstraction where a relationship between entities is treated as a higher-level single entity. When mapping aggregation, the "whole" (aggregate) relation includes the primary key of its "part" superclasses as foreign keys (DBMD UNIT - 1 and 2 Notes, p.49, p.50). This is distinct from directly mapping a higher-degree relationship but can sometimes be confused if the aggregate itself participates in further relationships.

The key is to ensure that the relational schema accurately captures all instances of the higher-degree relationship and any associated attributes.

UNIT-III
(Select one question from Q6 or Q7)

Question 6:

a) Describe the database creation using SQL with help of suitable examples. (8 Marks)

Answer:
Database creation using SQL primarily involves the `CREATE TABLE` statement, which is part of SQL's Data Definition Language (DDL). This statement defines a new base table, its columns, the data type for each column, and various constraints.

Syntax of `CREATE TABLE` (Simplified):

```
CREATE TABLE table_name (
    column_name_1 data_type [column_constraints],
    column_name_2 data_type [column_constraints],
    ...
    [table_constraints]
);
```

Key Components:

1. `table_name`: The name of the table to be created.
2. `column_name`: The name of a column in the table.
3. `data_type`: Specifies the type of data the column can hold. Common SQL data types include:

- **Numeric:** `INTEGER` (or `INT`), `SMALLINT`, `DECIMAL(p,s)`, `NUMERIC(p,s)`, `FLOAT`, `REAL`.
- **String:** `CHAR(n)` (fixed-length), `VARCHAR(n)` (variable-length).
- **Date/Time:** `DATE`, `TIME`, `TIMESTAMP`, `INTERVAL`.
- **Binary:** `BIT(n)`, `BIT VARYING(n)`.
- (Vendors may add more types or have variations, e.g., `TEXT`, `BLOB`, `CLOB`).

4. **`column_constraints`**: Apply to individual columns.
   - `NOT NULL`: Ensures the column cannot have a NULL value.
   - `UNIQUE`: Ensures all values in the column are unique across rows.
   - `PRIMARY KEY`: A shorthand for `NOT NULL` and `UNIQUE`, identifying the column as the primary key.
   - `CHECK (condition)`: Ensures values satisfy a specific condition.
   - `DEFAULT default_value`: Assigns a default value if none is provided during insertion.
   - `REFERENCES referenced_table(referenced_column)`: Defines a foreign key.
5. **`table_constraints`**: Apply to one or more columns, defined separately.
   - `PRIMARY KEY (column1, column2, ...)`: Defines a composite primary key.
   - `UNIQUE (column1, column2, ...)`: Defines a composite unique constraint.
   - `FOREIGN KEY (column_list) REFERENCES referenced_table(referenced_column_list) [ON DELETE action] [ON UPDATE action]`: Defines a foreign key, potentially composite, and specifies referential triggered actions (e.g., `CASCADE`, `SET NULL`, `RESTRICT`).
   - `CHECK (condition)`: A condition involving multiple columns.

Example: Creating Tables for a Medical System
(Based on Figure 10.1a, Box 1, Box 2, Box 3 in notes)

1. Patient Table:

```
CREATE TABLE Patient (
    Pat_p_alpha CHAR(2) NOT NULL,  -- Assuming part of PK
    Pat_p_num CHAR(5) NOT NULL,    -- Assuming part of PK
    Pat_name VARCHAR(100) NOT NULL,
    Pat_gender CHAR(1) CHECK (Pat_gender IN ('M', 'F')),
    Pat_age SMALLINT CHECK (Pat_age >= 0 AND Pat_age <= 120), -- Example age
constraint
    Pat_admit_date DATE,
    Pat_wing CHAR(1),
    Pat_room_num INT,
    Pat_bed CHAR(1) CHECK (Pat_bed IN ('A', 'B')),
    PRIMARY KEY (Pat_p_alpha, Pat_p_num)
);
```

2. Medication Table:

```
CREATE TABLE Medication (
    Med_code CHAR(5) PRIMARY KEY,
    Med_name VARCHAR(100) NOT NULL UNIQUE,
    Med_unit_price DECIMAL(5,2) CHECK (Med_unit_price > 0),
    Med_qty_onhand INT DEFAULT 0,
    Med_qty_onorder INT DEFAULT 0,
    CONSTRAINT chk_med_stock CHECK ((Med_qty_onhand + Med_qty_onorder) BETWEEN 0
AND 5000) -- Example constraint
);
```

3. Order Table (linking Patient and Medication):

```
CREATE TABLE Orders (
    Ord_rx_num CHAR(13) PRIMARY KEY,
    Ord_pat_p_alpha CHAR(2) NOT NULL,
    Ord_pat_p_num CHAR(5) NOT NULL,
    Ord_med_code CHAR(5) NOT NULL,
    Ord_dosage SMALLINT DEFAULT 1 CHECK (Ord_dosage IN (1, 2, 3)),
    Ord_freq SMALLINT DEFAULT 1 CHECK (Ord_freq IN (1, 2, 3)),
    FOREIGN KEY (Ord_pat_p_alpha, Ord_pat_p_num) REFERENCES Patient(Pat_p_alpha,
Pat_p_num)
        ON DELETE CASCADE ON UPDATE CASCADE, -- If patient is deleted, their orders
are deleted
    FOREIGN KEY (Ord_med_code) REFERENCES Medication(Med_code)
        ON DELETE RESTRICT ON UPDATE RESTRICT -- Prevent deleting/updating
medication if orders exist
);
```

This example demonstrates creating tables with various data types, primary keys (simple and composite), foreign keys with referential actions, CHECK constraints, and DEFAULT values.

(Based on: DBMD UNIT - 3 and 4 Notes, p.1-8)

---

b) Explain briefly DDL and DML with syntax and suitable examples. (7 Marks)

Answer:

DDL (Data Definition Language):
DDL statements are used to define, modify, and remove database structures (schemas) and objects. They don't manipulate the data itself but the containers and definitions for the data.

- **Key DDL Commands:**
    1. `CREATE`: Used to create new database objects like tables, views, indexes, schemas, domains.

- **Syntax (Table):** `CREATE TABLE table_name (column1 datatype constraints, ...);`
- **Example:** `CREATE TABLE Employees ( EmpID INT PRIMARY KEY, EmpName VARCHAR(100), Salary DECIMAL(10,2) );`

2. `ALTER`: Used to modify the structure of existing database objects.
   - **Syntax (Table - Add Column):** `ALTER TABLE table_name ADD column_name datatype constraints;`
   - **Syntax (Table - Drop Column):** `ALTER TABLE table_name DROP COLUMN column_name;`
   - **Syntax (Table - Modify Column):** `ALTER TABLE table_name MODIFY COLUMN column_name new_datatype;` (Syntax varies by DBMS)
   - **Example:** `ALTER TABLE Employees ADD Department VARCHAR(50);`
3. `DROP`: Used to delete existing database objects.
   - **Syntax (Table):** `DROP TABLE table_name [CASCADE | RESTRICT];`
   - **Example:** `DROP TABLE Employees;` (This deletes the table structure and all its data).
4. `TRUNCATE`: (Often considered DDL) Removes all rows from a table quickly, but the table structure remains. It's usually faster than `DELETE` without a `WHERE` clause as it typically doesn't log individual row deletions.
   - **Syntax (Table):** `TRUNCATE TABLE table_name;`
   - **Example:** `TRUNCATE TABLE LogEntries;`

DML (Data Manipulation Language):
DML statements are used to retrieve, insert, update, and delete data within database tables.

- **Key DML Commands:**
  1. `SELECT`: Used to retrieve data from one or more tables.
     - **Syntax:** `SELECT column_list FROM table_name [WHERE condition] [ORDER BY column_name];`
     - **Example:** `SELECT EmpName, Salary FROM Employees WHERE Salary > 50000;`
  2. `INSERT`: Used to add new rows (records) into a table.
     - **Syntax:** `INSERT INTO table_name (column1, column2, ...) VALUES (value1, value2, ...);`
     - **Example:** `INSERT INTO Employees (EmpID, EmpName, Salary, Department) VALUES (101, 'Alice Smith', 60000, 'HR');`
  3. `UPDATE`: Used to modify existing data in a table.
     - **Syntax:** `UPDATE table_name SET column1 = value1, column2 = value2, ... [WHERE condition];`
     - **Example:** `UPDATE Employees SET Salary = 65000 WHERE EmpID = 101;`

4. `DELETE`: Used to remove existing rows from a table.
    - **Syntax:** `DELETE FROM table_name [WHERE condition];`
    - **Example:** `DELETE FROM Employees WHERE EmpID = 101;` (If `WHERE` is omitted, all rows are deleted).

(Based on: DBMD UNIT - 1 and 2 Notes, p.4; DBMD UNIT - 3 and 4 Notes, p.1, p.8, p.12)

Question 7:

a) Describe the cursor and type of cursor. What is the need of cursor in database programming? (8 Marks)

Answer:
Cursor:
A cursor in database programming is a control structure that enables traversal over the records in a database. More specifically, it's a pointer to a specific row within a result set returned by an SQL query. Cursors facilitate row-by-row processing of the result set within an application program, bridging the gap between set-oriented SQL and row-oriented procedural programming languages.

Need for Cursors in Database Programming: (DBMD UNIT - 3 and 4 Notes, p.18)
SQL is a set-oriented language; its operations (like `SELECT`) act on and return sets of rows. Host programming languages (like C, Java, Python) are typically record-oriented or object-oriented, processing data one record/object at a time. This difference creates an **impedance mismatch**.

When an SQL query embedded in a host language program returns multiple rows, the host language usually doesn't have a native data structure to handle this entire set at once directly from the SQL engine. Cursors bridge this gap by providing a mechanism for the application program to:

1. **Define a set of rows** (the result of an SQL query).
2. **Iterate through this set**, retrieving one row at a time into host language variables for processing.
3. Potentially **update or delete the row** currently pointed to by the cursor (for updatable cursors).

Operations on Cursors: (DBMD UNIT - 3 and 4 Notes, p.18)

1. **DECLARE:** Defines the cursor by associating it with a `SELECT` statement and giving it a name.
   `DECLARE cursor_name [INSENSITIVE] [SCROLL] CURSOR FOR SELECT_statement [ORDER BY ...] [FOR READ ONLY | FOR UPDATE [OF column_list]];`
2. **OPEN:** Executes the query associated with the cursor, populates the result set, and positions the cursor *before* the first row.
   `OPEN cursor_name;`
3. **FETCH:** Retrieves the next (or specified) row from the result set and loads its data into host language variables. It also advances the cursor.

`FETCH [NEXT | PRIOR | FIRST | LAST | ABSOLUTE n | RELATIVE n] FROM cursor_name INTO :host_variable_list;`

4. **UPDATE (positioned):** Modifies the row currently pointed to by an updatable cursor. `UPDATE table_name SET ... WHERE CURRENT OF cursor_name;`
5. **DELETE (positioned):** Deletes the row currently pointed to by an updatable cursor. `DELETE FROM table_name WHERE CURRENT OF cursor_name;`
6. **CLOSE:** Deactivates the cursor and releases the resources (like the result set) associated with it. `CLOSE cursor_name;`

Types of Cursors (Based on Properties and Behavior): (DBMD UNIT - 3 and 4 Notes, p.19-20)

1. **Read-Only Cursor (Default in many cases):**
   - **Purpose:** Allows only fetching (retrieving) data. No updates or deletions through the cursor are permitted.
   - **Declaration:** Often the default, or explicitly `FOR READ ONLY`.
2. **Updatable Cursor:**
   - **Purpose:** Allows fetching data and also modifying (`UPDATE ... WHERE CURRENT OF`) or deleting (`DELETE ... WHERE CURRENT OF`) the row currently pointed to by the cursor.
   - **Declaration:** `FOR UPDATE [OF column_list]`. The `OF column_list` is optional and specifies which columns can be updated.
   - **Restrictions:** The query defining an updatable cursor must typically be simple (e.g., based on a single table, no aggregates, `GROUP BY`, `DISTINCT`).
3. **Forward-Only Cursor (Non-Scrollable):**
   - **Purpose:** The default type. Rows can only be fetched sequentially from the first to the last. No backward movement or jumping to specific rows is allowed.
   - **Behavior:** Generally most efficient as it requires fewer resources from the DBMS.
4. **Scrollable Cursor:**
   - **Purpose:** Allows flexible movement within the result set beyond just fetching the next row.
   - **Declaration:** `SCROLL CURSOR`.
   - **Fetch options:** `NEXT`, `PRIOR`, `FIRST`, `LAST`, `ABSOLUTE n` (go to n-th row), `RELATIVE n` (move n rows from current).
5. **Insensitive Cursor (Snapshot Cursor):**
   - **Purpose:** The cursor operates on a temporary copy (snapshot) of the data as it existed when the cursor was opened.
   - **Declaration:** `INSENSITIVE CURSOR`.
   - **Behavior:** Changes made to the underlying base tables by other transactions (or even the same transaction outside the cursor) after the cursor is opened are *not* visible to this cursor. Provides

a static view, good for consistency but data might become stale.

6. **Sensitive Cursor:**
    - **Purpose:** Attempts to reflect changes made to the underlying data by other transactions after the cursor was opened. The degree of sensitivity can vary.
    - **Behavior:** More complex for the DBMS to implement. The exact behavior (which changes are visible and when) can be DBMS-dependent. Changes might make the current row invalid or cause rows to appear/disappear from the result set.
7. **Keyset-Driven Cursor (A type of sensitive cursor):**
    - **Purpose:** When opened, the set of keys for the qualifying rows is fixed and stored. The cursor uses these keys to fetch rows.
    - **Behavior:** Changes to non-key values of rows in the keyset are visible. Rows deleted by others will appear as "holes." New rows inserted by others that would qualify are typically not seen.

(Note: Static Cursor and Dynamic Cursor mentioned on p.20 of notes are broader terms often mapping to Insensitive and highly Sensitive cursors respectively.)

---

b) What is database trigger? Explain the types of Trigger. (7 Marks)

Answer:
Database Trigger:
A database trigger is a procedural code (a block of SQL statements or a call to a stored procedure) that is automatically executed by the DBMS in response to certain Data Manipulation Language (DML) events (like `INSERT`, `UPDATE`, `DELETE`) or sometimes Data Definition Language (DDL) events (`CREATE`, `ALTER`, `DROP`) on a specified table or view. Triggers are stored in the database and are managed by the DBMS.

ECA Model (Event-Condition-Action): (DBMD UNIT - 3 and 4 Notes, p.21)
Triggers typically follow an Event-Condition-Action model:

- **Event:** The DML or DDL operation that causes the trigger to fire (e.g., an `INSERT` on the `Employees` table).
- **Condition (Optional):** A Boolean expression. If specified, the trigger's action is executed only if the condition evaluates to true.
- **Action:** The procedural code (SQL block or procedure call) that is executed when the event occurs and the condition (if any) is met.

Syntax (Simplified Generic): (DBMD UNIT - 3 and 4 Notes, p.21)

```
CREATE [OR REPLACE] TRIGGER trigger_name
{BEFORE | AFTER} {INSERT | DELETE | UPDATE [OF column_list]}
ON table_name
[FOR EACH ROW]
```

```
[WHEN (condition)]
BEGIN
    -- Trigger action (SQL statements)
END;
```

Types of Triggers:

1. **Row-Level Trigger vs. Statement-Level Trigger:**
    - **Row-Level Trigger (`FOR EACH ROW`):**
        - Fires once for *each affected row* by the DML statement.
        - Can access the old and new values of the row being processed using special correlation names (often `:OLD` and `:NEW` or `inserted`/`deleted` pseudo-tables).
        - **Example:** An auditing trigger that logs details of every row updated in an `Employees` table.
    - **Statement-Level Trigger (Default if `FOR EACH ROW` is omitted):**
        - Fires once for the *entire DML statement*, regardless of how many rows are affected (even if zero rows are affected).
        - Cannot directly access `:OLD` or `:NEW` row values because it acts on the statement as a whole.
        - **Example:** A trigger that sends an alert if any attempt is made to delete records from a critical table.
2. **Timing of Trigger Firing (BEFORE vs. AFTER):**
    - **BEFORE Trigger:**
        - The trigger action executes *before* the triggering DML statement is actually performed on the table.
        - **Uses:**
            - Validating input data or modifying new data before it's written (e.g., converting a value to uppercase).
            - Preventing certain operations based on conditions.
            - Deriving values for columns before an insert or update.
    - **AFTER Trigger:**
        - The trigger action executes *after* the triggering DML statement has completed and all related constraints have been checked.
        - **Uses:**
            - Auditing changes (logging what happened).
            - Enforcing complex referential integrity or business rules that cannot be handled by standard constraints.
            - Propagating changes to other related tables.

3. **INSTEAD OF Triggers:** (DBMD UNIT - 3 and 4 Notes, p.21)
    - **Purpose:** Used primarily with views, especially complex views that are not inherently updatable (e.g., views involving joins, aggregates, `GROUP BY`, `DISTINCT`).
    - **Behavior:** When a DML operation (`INSERT`, `UPDATE`, `DELETE`) is attempted on the view, the `INSTEAD OF` trigger fires *instead* of the DML operation on the view. The trigger's code then defines how to translate this operation into appropriate DML operations on the underlying base tables.
    - **Example:** An `INSTEAD OF INSERT` trigger on a view joining `EMPLOYEE` and `DEPARTMENT` tables might insert data into both base tables correctly.
4. **DDL Triggers:** (DBMD UNIT - 3 and 4 Notes, p.22)
    - **Purpose:** Fire in response to DDL events (e.g., `CREATE TABLE`, `ALTER TABLE`, `DROP TABLE`, `CREATE INDEX`).
    - **Behavior:** Used for auditing schema changes, enforcing naming conventions, preventing certain DDL operations, or automating related tasks.
    - **Availability:** Not part of standard SQL for all DDL events but supported by some DBMSs (e.g., SQL Server, Oracle).
5. **Logon/Logoff Triggers (System-Level Triggers):** (DBMD UNIT - 3 and 4 Notes, p.22)
    - **Purpose:** Fire when a user session connects to (`LOGON`) or disconnects from (`LOGOFF`) the database.
    - **Behavior:** Used for auditing user sessions, setting session-specific parameters, or restricting connections based on time or location.
    - **Availability:** DBMS-specific.
6. **Database Event Triggers (Server-Level Triggers):** (DBMD UNIT - 3 and 4 Notes, p.22)
    - **Purpose:** Fire in response to database-level events such as startup, shutdown, or specific errors occurring at the server level.
    - **Behavior:** Used for administrative tasks, custom error logging, or initiating recovery procedures.
    - **Availability:** DBMS-specific.
7. **Compound Triggers (Oracle specific):** (DBMD UNIT - 3 and 4 Notes, p.22)
    - **Purpose:** Allows defining actions for multiple timing points (`BEFORE STATEMENT`, `BEFORE EACH ROW`, `AFTER EACH ROW`, `AFTER STATEMENT`) for a single DML event on a table within a single trigger body.
    - **Behavior:** Can simplify logic and share variables across different timing points for the same triggering event.

Triggers are powerful tools for maintaining data integrity, enforcing business rules, and automating actions, but they can also add complexity and overhead if not designed and used carefully.

UNIT-IV
(Select one question from Q8 or Q9)

Question 8:

a) Explain the Indexing and its methods with the help of suitable examples. (8 Marks)

Answer:
Indexing:
Indexing in a database is a technique to speed up the retrieval of rows from a table. An index is a separate data structure (often a B+-tree or hash table) that stores a copy of one or more columns (the index key) from a table, along with pointers (rowids or physical addresses) to the actual data rows. When a query filters or sorts by an indexed column, the DBMS can use the index to quickly locate the relevant rows without scanning the entire table (full table scan), significantly improving query performance, especially for large tables.

Why Indexing?
Without an index, retrieving data based on a condition requires a full table scan, where the DBMS reads every row to check if it matches. This is inefficient for large tables. Indexes provide a shortcut.

Methods/Types of Indexing:

1. **Primary Index (often Clustered):**
    - **Description:** An index whose search key also specifies the sequential order of the data file. The data file is ordered by this key.
    - **Example:** If `EmployeeID` is the primary key and a primary index is created on it, the employee records in the table might be physically stored in `EmployeeID` order.
    - **Note:** A table can have at most one primary (and thus one clustered) index.
2. **Clustered Index:**
    - **Description:** The physical order of data rows in the table is the same as the order of the index key values.
    - **Example:** `CREATE CLUSTERED INDEX IX_Orders_OrderDate ON Orders(OrderDate);` If this is created, rows in the `Orders` table are physically sorted by `OrderDate`.
    - **Benefit:** Very efficient for range queries on the clustering key and for retrieving rows in the key's order.
3. **Secondary Index (Non-Clustered Index):**
    - **Description:** An index whose search key specifies an order different from the sequential order of the file. The data rows are not physically ordered according to this index key. The index entries point to the data rows.
    - **Example:** `CREATE INDEX IX_Employees_LastName ON Employees(LastName);` Employee records are not physically sorted by `LastName`, but this index allows quick lookups by `LastName`.

- **Note:** A table can have multiple secondary indexes.

4. **B+-Tree Index:**
   - **Description:** The most common index structure. It's a balanced tree where all leaf nodes are at the same depth. Leaf nodes contain (key value, pointer to data record/rowid) pairs and are linked sequentially, allowing efficient range searching and equality searches.
   - **Example:** Most default indexes created in relational databases are B+-tree indexes. `CREATE INDEX IX_Products_Price ON Products(Price);` would likely create a B+-tree on `Price`.
5. **Hash Index:**
   - **Description:** Uses a hash function to compute the address of the bucket/page where the index entry (and possibly data record) for a key value is stored.
   - **Example:** If an index is created on `CustomerEmail` using a hash structure, searching for `WHERE CustomerEmail = 'test@example.com'` would be very fast.
   - **Benefit/Drawback:** Very fast for exact equality searches. Not suitable for range queries (e.g., `Price > 100`).
6. **Unique Index:**
   - **Description:** Enforces that no two rows in the table can have the same value for the indexed column(s). Primary key indexes are always unique.
   - **Example:** `CREATE UNIQUE INDEX UQ_Employees_Email ON Employees(Email);` ensures no two employees have the same email.
7. **Composite Index (Multi-column Index):**
   - **Description:** An index created on two or more columns. The order of columns in the index definition is crucial for query performance.
   - **Example:** `CREATE INDEX IX_Orders_CustDate ON Orders(CustomerID, OrderDate);` This index is useful for queries filtering on `CustomerID` or on `CustomerID AND OrderDate`.
8. **Covering Index:**
   - **Description:** A non-clustered index that contains all the columns required to satisfy a query (both in `SELECT` list and `WHERE` clause).
   - **Example:** For query `SELECT OrderDate, OrderAmount FROM Orders WHERE CustomerID = 123;`, an index `CREATE INDEX IX_Cover_Orders ON Orders(CustomerID, OrderDate, OrderAmount);` would be a covering index. The query can be answered entirely from the index (index-only scan) without accessing the table.

(Based on: DBMD UNIT - 3 and 4 Notes, p.23-27)

---

b) Discuss about the database security used in the database modelling. (7 Marks)

Answer:
Integrating database security considerations into the database modeling process is crucial for building

secure and robust systems. Security should not be an afterthought but an integral part from requirements gathering through physical design.

Security Considerations During Database Modeling Phases:

1. **Conceptual Level (ER/EER Modeling & Requirements):**
    - **Identify Sensitive Data:** During requirements gathering, clearly identify which entities and attributes contain sensitive information that requires protection. Examples include Personally Identifiable Information (PII) like social security numbers, financial data like salaries or credit card numbers, and personal health information.
    - **Define Access Privileges Conceptually (User Roles):** Understand and document the different roles of users who will interact with the system (e.g., HR manager, sales representative, clerk, customer). For each role, define what data they *should* be able to see (read), create, modify, or delete. This doesn't mean drawing security constraints directly in the ERD but documenting these requirements alongside it.
    - **Consider Views for Abstraction and Security:** If certain user groups only need to see a subset of attributes from an entity or aggregated data, this can be noted conceptually. This leads to the creation of database views later, which act as a security layer. For example, a view might show average salaries by department but not individual salaries.
    - **Data Ownership:** Clarify who "owns" different pieces of data or entities. Data ownership often dictates who has the authority to grant access permissions to that data.
2. **Logical Level (Relational Schema Design):**
    - **View Design for Security:** Based on conceptual requirements, design specific database views.
        - **Column-level security:** Views can be created to select only a subset of columns from a base table, hiding sensitive columns from certain users.
        - **Row-level security (Value-Dependent Access Control):** Views can use a `WHERE` clause to filter rows based on user identity or attributes. For example, a sales manager might only see orders for their specific region through a view.
    - **Granular Privilege Planning:** Plan for the types of SQL privileges (`SELECT`, `INSERT`, `UPDATE`, `DELETE`, `REFERENCES`) that will be needed on tables and views for different user roles. This informs the DCL (GRANT/REVOKE) statements that will be implemented.
    - **Normalization and Integrity:** While not directly security, proper normalization helps maintain data integrity, which is a facet of security (preventing unauthorized or inconsistent data modification). Foreign key constraints also play a role in maintaining valid relationships.
3. **Physical Level (Implementation in DBMS - Informed by Modeling):**
    - The conceptual and logical security decisions directly influence how security is implemented physically:
        - **GRANT/REVOKE (Discretionary Access Control - DAC):** Implement the access privileges defined during modeling.

- **Role-Based Access Control (RBAC):** Create roles (defined conceptually) and grant privileges to roles, then assign users to roles.
- **Stored Procedures for Controlled Access:** Sensitive DML operations can be encapsulated in stored procedures. Users are granted `EXECUTE` permission on the procedure but not direct DML access to tables.
- **Triggers for Auditing:** Implement triggers (planned during logical design if complex auditing rules are needed) to log access to sensitive data.
- **Encryption:** If modeling identified highly sensitive data (e.g., credit card numbers), plan for column-level encryption or Transparent Data Encryption (TDE) at the physical level.

By considering security at each stage of database modeling, organizations can ensure that access controls are aligned with business requirements, sensitive data is appropriately protected, and the principle of least privilege is enforced.

(Based on: DBMD UNIT - 3 and 4 Notes, p.31-32)

Question 9:

Write short notes on (5x3 = 15 Marks)

a) Clustering

Answer:
Clustering in the context of databases refers to the physical storage of related data records close together on disk, ideally in the same or adjacent disk blocks. The primary goal of clustering is to minimize Disk I/O operations when accessing these related records together, thereby improving query performance.

How it Works:
Clustering is typically achieved by organizing data rows based on the values of one or more columns, known as the **clustering key**.

Types of Clustering: (DBMD UNIT - 3 and 4 Notes, p.28)

1. **Intra-Table Clustering (Clustered Index):**
    - **Description:** The physical storage order of data rows within a single table matches the logical order of an index created on that table (the clustered index).
    - **Impact:** A table can have at most one clustered index because the data rows themselves are sorted.
    - **Benefit:** Extremely efficient for range queries on the clustering key and for retrieving rows in the key's sorted order. Often, the primary key is chosen as the clustering key.

- **Example:** If an `Orders` table has a clustered index on `OrderDate`, the order records will be physically stored on disk sorted by their `OrderDate`.

2. **Inter-Table Clustering (Co-clustering / Multi-Table Clustering):**
    - **Description:** Physically interleaving rows from two or more related tables on disk, based on their common join key (e.g., a PK-FK relationship).
    - **Impact:** Speeds up frequent joins between these tables significantly by reducing the I/O needed to fetch related rows from different tables, as they are already stored close together.
    - **Example:** Storing `OrderItem` records physically near their parent `Order` record. When an `Order` is fetched, its related `OrderItems` are likely in the same or nearby disk blocks.

Benefits of Clustering:

- **Reduced Disk I/O:** The main advantage, leading to faster query performance for operations that access clustered data.
- **Improved Performance for Joins:** Especially with inter-table clustering.
- **Efficient Range Queries:** For intra-table clustering on the range attribute.

Drawbacks of Clustering:

- **Maintenance Overhead:** `INSERT`, `DELETE`, `UPDATE` operations can be slower as the physical order might need to be maintained, potentially causing page splits or data movement.
- **Only One Clustered Index per Table:** For intra-table clustering.
- **Full Table Scans:** Might be slightly slower if data pages are not as densely packed due to maintaining cluster order.

(Based on: DBMD UNIT - 3 and 4 Notes, p.23, p.28-29)

---

b) De-normalization

Answer:
Denormalization is the process of intentionally introducing controlled redundancy into a relational database schema by adding duplicate data or grouping data that was separated in the normalization process. It is the reverse of normalization. The primary goal of denormalization is to improve read performance (query speed) for specific, critical queries by reducing the number of joins required or by pre-calculating values.

Rationale/Use Case: (DBMD UNIT - 3 and 4 Notes, p.29-30)
Highly normalized schemas are excellent for data integrity and minimizing update anomalies, but they can sometimes lead to slow query performance due to the need for many joins to retrieve related information. Denormalization is a performance optimization technique applied selectively when indexing and query tuning are insufficient.

Types of Denormalization / Techniques: (DBMD UNIT - 3 and 4 Notes, p.30)

1. **Pre-joining Tables:** Adding attributes from a "one-side" table to a frequently joined "many-side" table to avoid a join.
    - **Example:** Adding `DepartmentName` (from `DEPARTMENT` table) to the `EMPLOYEE` table. Queries needing employee name and department name can then access only the `EMPLOYEE` table.
2. **Storing Derived/Calculated Values:** Storing pre-computed values (e.g., totals, averages) that are expensive to calculate on-the-fly.
    - **Example:** Storing `OrderTotal` in an `ORDERS` table instead of calculating it by summing `OrderItem` amounts each time.
3. **Combining Tables:** Merging tables that have a 1:1 relationship or a very tight, frequently accessed 1:N relationship where the "many" side has few records per "one" side.
4. **Repeating Groups (Limited Use):** Using multiple columns for a fixed, small number of similar attributes (e.g., `Phone1`, `Phone2`) instead of a separate related table. Generally discouraged due to inflexibility but sometimes used for very specific cases.
5. **Creating Reporting Tables/Data Marts:** Building separate, denormalized tables specifically for analytical queries and reporting, populated from the normalized operational database.

Trade-offs:

- **Benefit:** Faster read/query performance for targeted queries.
- **Cost/Drawback:**
    - **Increased Storage:** Due to data redundancy.
    - **Data Inconsistency Risk:** Duplicated data must be kept synchronized. `UPDATE`, `INSERT`, `DELETE` operations become more complex and require careful management (e.g., using triggers or application logic) to maintain consistency.
    - **Slower Update Performance:** Updates might need to modify data in multiple places.
    - **More Complex DML/Application Logic:** To handle redundant data.

Guideline: Apply denormalization selectively and judiciously *after* normalization and only when there's a clear performance bottleneck that cannot be resolved by other means like indexing or query optimization.

(Based on: DBMD UNIT - 3 and 4 Notes, p.29-30)

---

c) Database Tuning

Answer:
Database tuning is the systematic process of optimizing various aspects of a database system to improve its performance, meet user requirements, and achieve service level agreements. It's an iterative process aimed at identifying and resolving performance bottlenecks.

Key Areas of Database Tuning: (DBMD UNIT - 3 and 4 Notes, p.22, p.30-31)

1. **Tuning the Conceptual Schema (Logical Design):**
    - **Normalization/Denormalization:** Evaluating if the current level of normalization is appropriate. Sometimes, denormalizing specific parts can improve read performance for critical queries (as discussed above). Conversely, further normalization might be needed if update anomalies are an issue.
    - **Vertical Partitioning:** Splitting a table's columns into multiple tables if some columns are accessed much more frequently than others.
    - **Horizontal Partitioning:** Splitting a table's rows into multiple tables (or partitions within a table) based on criteria like date range or region, often to improve manageability and query performance on subsets of data.
2. **Tuning Queries and Views:**
    - **Rewriting SQL Queries:** Optimizing inefficient SQL statements by avoiding unnecessary joins, using sargable predicates (conditions that can efficiently use indexes), simplifying complex logic, or using more efficient constructs.
    - **Optimizing View Definitions:** Ensuring views are defined efficiently, especially if they are complex or frequently queried.
    - **Understanding and Influencing Query Optimizer:** Analyzing query execution plans to understand how the DBMS is processing queries. Using hints (if necessary and supported) or ensuring statistics are up-to-date to guide the optimizer.
3. **Tuning Physical Design (Indexing and Storage):**
    - **Index Selection and Management:**
        - Creating appropriate indexes (B+-tree, hash, composite, covering, clustered) on columns frequently used in `WHERE`, `JOIN`, `ORDER BY`, and `GROUP BY` clauses.
        - Dropping unused or ineffective indexes that add overhead to DML operations.
        - Rebuilding or reorganizing indexes periodically.
    - **Clustering:** Deciding if and how to cluster data (intra-table or inter-table) to reduce I/O for specific access patterns.
    - **File Organization and Placement:** Optimizing how data files are stored on disk.
    - **Disk Space Management:** Ensuring adequate disk space and managing fragmentation.
4. **Tuning Application Code:**
    - Optimizing how applications interact with the database (e.g., reducing round trips, using batch updates, efficient connection management).
    - Choosing appropriate data access methods (e.g., Embedded SQL vs. API, ORM settings).
5. **Tuning DBMS Parameters:**
    - Adjusting DBMS configuration parameters related to memory allocation (e.g., buffer pool size), I/O, concurrency control, logging, etc. This is highly DBMS-specific.

6. **Tuning Hardware and Operating System:**
    - Ensuring sufficient CPU, memory, and fast I/O subsystems.
    - Optimizing operating system settings for database workloads.

Iterative Process: (DBMD UNIT - 3 and 4 Notes, p.22)
Database tuning is not a one-time task but an ongoing, iterative cycle:

1. **Monitor:** Continuously monitor database performance using DBMS tools, OS utilities, and application performance monitoring.
2. **Identify Bottlenecks:** Analyze monitoring data to pinpoint areas of poor performance (e.g., slow queries, high I/O, CPU contention).
3. **Diagnose:** Determine the root cause of the bottlenecks.
4. **Implement Changes:** Apply tuning measures (e.g., add an index, rewrite a query, change a parameter).
5. **Measure:** Evaluate the impact of the changes. If performance improves, keep the change; otherwise, revert or try another approach.

Workload Analysis: (DBMD UNIT - 3 and 4 Notes, p.22)
A critical first step in tuning is understanding the database workload:

- Types and frequencies of queries and updates.
- Performance goals for each type of operation.
- Accessed relations, attributes, selection/join conditions, and their selectivity.

(Based on: DBMD UNIT - 3 and 4 Notes, p.22, p.29-31)
