**Unit I: Introduction to Database Management Systems and ER Model**

**1. Introduction to DBMS**

* Defines a Database Management System (DBMS) as a collection of interrelated data and programs to manage and retrieve data effectively.
* The main goals of DBMS include efficient data storage and retrieval, providing a structured environment for data manipulation, and ensuring data safety.

**2. Purpose of Database Systems**

* Discusses the evolution from file processing systems to DBMS to overcome redundancy and improve data integrity.
* Aims to provide an organized, efficient system for data storage and management that supports various applications.

**3. Advantages of DBMS over File Processing Systems**

* Highlights key benefits, including reduced data redundancy, improved data sharing, enforcement of standards, and data security, alongside challenges like complexity and potential cost increases.

**4. Database-System Applications**

* Covers DBMS applications across fields such as banking, airlines, telecommunication, and education, emphasizing their role in managing large volumes of data.

**5. View of Data**

* Explains the concept of data abstraction through three levels: physical, logical, and view levels, to separate user applications from complex data structures.

**6. Data Abstraction**

* Provides a way to represent data independently at different levels, facilitating easier data management and retrieval while hiding implementation details.

**7. Database Languages**

* Outlines the types of languages used in DBMS, including Data Definition Language (DDL), Data Manipulation Language (DML), and Data Control Language (DCL) for tasks like defining, manipulating, and securing data.

**8. Database System Structure**

* Describes the architecture of DBMS, including components like the storage manager, query processor, and transaction manager, which support data handling and query execution.

**9. Data Models**

* Introduces the various data models, such as the relational model, entity-relationship model, and object-based model, which define how data is organized, related, and manipulated in a DBMSndence\*\*
* Emphasizes the separation between application programs and data to protect database structures from application changes, achieved through logical and physical data independence.

**11. Database Users**

* Categorizes database users, including naïve users, application programmers, and specialized users, each interacting with the DBMS through different interfaces for tasks like query execution and application development .

**Databasd ER Model**

**1. Database Design and ER Model**

* The ER Model focuses on the conceptual representation of data, identifying entities, relationships, and attributes to structure the database logically.

**2. Entity and Entity Sets**

* Defines entities as objects distinguishable from other objects (e.g., a student), with attributes describing them. An entity set is a collection of similar entities .

**3. Relationships and Relets**

* Relationships link entities and are stored as relationship sets (e.g., "works\_for" relationship between "Employee" and "Department").

**4. Constraints and Keys**

* Constraints ensure data validity (e.g., cardinality and participation), while keys uniquely identify entities or tuples in a database, supporting database integrity.

**5. ER Diagram and Design Process**

* ER diagrams represent data graphically, simplifying the database design process by illustrating entities, attributes, and relationships. Extended ER features like generalization, specialization, and aggregation add layers of complexity .

**Unit II: SQL and PL/SQL**

**1. SQL - Structured Query Language**

* **Definition**: SQL is a standardized language for managing relational databases, providing methods for querying, updating, and managing data.
* **Purpose**: SQL is designed to interact with data within a database by defining, manipulating, and retrieving it efficiently.
* **Main Points**:
  + SQL is case-insensitive and used across various RDBMS platforms, including Oracle, MySQL, and SQL Server.
  + Includes categories like DDL (Data Definition Language), DML (Data Manipulation Language), DCL (Data Control Language), and TCL (Transaction Control Language) for distinct data operations.
* **Significance**: SQL is the backbone of relational databases, making it fundamental for any database application, from simple queries to complex data transformations.

**2. SQL Data Types and Literals**

* **Definition**: Data types in SQL categorize the type of data stored, such as integers, strings, and dates.
* **Purpose**: Enforces structure in the database by defining the kind of data that each column can store, aiding in consistency and data validation.
* **Main Points**:
  + **Numeric**: Supports integers (INT, SMALLINT) and real numbers (REAL, FLOAT).
  + **Character**: Fixed (CHAR) and variable-length (VARCHAR) strings.
  + **Date and Time**: DATE, TIME, and TIMESTAMP for temporal data.
* **Significance**: Ensures data integrity by controlling and validating inputs, which is critical for maintaining a clean and reliable dataset.

**3. DDL, DML, DCL, and TCL**

* **Definition**: These are the primary categories of SQL commands.
* **Purpose**: Each category manages different database functionalities—from defining structures (DDL) to controlling transactions (TCL).
* **Main Points**:
  + **DDL**: Commands like CREATE, ALTER, DROP define database schema.
  + **DML**: Commands like SELECT, INSERT, UPDATE handle data manipulation.
  + **DCL**: Commands such as GRANT and REVOKE manage access permissions.
  + **TCL**: Commands like COMMIT, ROLLBACK ensure data consistency.
* **Significance**: These command types provide complete control over the database, from setup to transaction management.

**4. Tables and Table Operations**

* **Definition**: Tables are the main structure in a database, storing rows and columns of related data.
* **Purpose**: Organizes data in a structured format, enabling efficient storage, retrieval, and manipulation.
* **Main Points**:
  + **Creating**: Syntax for creating tables with defined data types and constraints.
  + **Modifying and Deleting**: ALTER for changes, DROP for deletion.
* **Significance**: Tables serve as the fundamental data storage unit, making their creation and management essential for database operation.

**5. SQL DML Queries**

* **Definition**: DML queries are commands that allow data manipulation within tables.
* **Purpose**: Enables users to retrieve, insert, update, and delete data.
* **Main Points**:
  + **SELECT**: Retrieves data based on specified conditions.
  + **INSERT**: Adds new records.
  + **UPDATE and DELETE**: Modify or remove existing data.
* **Significance**: DML commands are crucial for dynamically interacting with and modifying the data, forming the core of data handling in applications.

**6. PL/SQL - Procedural Language/SQL**

* **Definition**: PL/SQL is Oracle’s procedural extension to SQL, combining SQL’s data handling with procedural programming constructs.
* **Purpose**: Enhances SQL capabilities by adding features like variables, loops, and conditionals, enabling complex operations and data manipulation.
* **Main Points**:
  + Supports **Stored Procedures** for modular programming.
  + **Cursors** manage query results, while **Triggers** automate actions based on events.
  + Control structures like loops and conditionals allow for procedural logic.
* **Significance**: PL/SQL extends SQL’s functionality, making it a powerful tool for complex database interactions and backend logic.

**7. Stored Procedures and Functions in PL/SQL**

* **Definition**: Stored procedures and functions are PL/SQL blocks that perform specific tasks and can be saved and reused.
* **Purpose**: Encapsulate and execute repetitive logic or calculations within the database to enhance reusability and efficiency.
* **Main Points**:
  + **Procedures** can perform actions but don’t return values.
  + **Functions** perform actions and return a single value.
* **Significance**: Reduces redundancy, centralizes logic, and improves database efficiency by avoiding repetitive coding.

**8. Cursors**

* **Definition**: A cursor is a pointer that enables row-by-row processing of SQL query results.
* **Purpose**: Allows for granular control over query results, essential for handling large datasets within procedural blocks.
* **Main Points**:
  + **Implicit Cursors** for simple SELECT statements.
  + **Explicit Cursors** for more complex queries that require detailed handling.
* **Significance**: Cursors facilitate detailed data manipulation, critical for applications that require precise record-level operations.

**9. Triggers**

* **Definition**: A trigger is a PL/SQL block that automatically executes in response to specified events on a table.
* **Purpose**: Automates tasks such as logging, enforcing rules, or auditing changes within the database.
* **Main Points**:
  + Triggered by **INSERT**, **UPDATE**, or **DELETE** events.
  + Ensures data consistency and integrity without manual intervention.
* **Significance**: Triggers provide a safeguard for data integrity and enable automatic responses to data changes, enhancing database reliability.

**10. Assertions, Roles, and Privileges**

* **Definition**: Assertions enforce rules; roles and privileges control access.
* **Purpose**: Ensures data integrity through constraints and controls user access to maintain security.
* **Main Points**:
  + **Assertions** enforce conditions on data.
  + **Roles and Privileges** define and grant access rights to users.
* **Significance**: These features protect data integrity and control user interactions, which is crucial for data security.

**Unit III: Relational Database Design**

**1. Relational Model - Basic Concepts**

* **Definition**: A relational model represents data as a collection of tables, where each table (or relation) consists of rows and columns.
* **Purpose**: Structures data for efficient organization and retrieval, providing a clear format for representing data in databases.
* **Main Points**:
  + Tables have unique names and consist of attributes (columns) and tuples (rows).
  + Relationships between tables are established through keys.
* **Significance**: Forms the foundation of relational databases, enabling structured data storage and retrieval.

**2. Attributes and Domains**

* **Definition**: Attributes represent the properties of an entity, and domains define the possible values that an attribute can hold.
* **Purpose**: Ensures data consistency by restricting attribute values to specified sets (domains).
* **Main Points**:
  + Attributes in a table have unique names, while domains specify valid data types (e.g., integers, strings).
  + Domains also support constraints for data validation.
* **Significance**: Attributes and domains maintain data integrity, crucial for database accuracy.

**3. CODD's Rules**

* **Definition**: A set of 12 rules defined by E.F. Codd to evaluate a system's adherence to relational model principles.
* **Purpose**: Serves as a benchmark for relational database management systems (RDBMS) to ensure compliance with relational concepts.
* **Main Points**:
  + Includes rules for data independence, integrity, and manipulation.
  + Ensures RDBMS functionalities align with relational theory.
* **Significance**: Codd's rules help verify whether a DBMS is truly relational, guiding database design best practices.

**4. Relational Integrity - Keys**

* **Definition**: Keys are attributes or sets of attributes that uniquely identify tuples in a table.
* **Purpose**: Ensures each record is uniquely identifiable, aiding in data retrieval and integrity.
* **Main Points**:
  + **Primary Key**: A unique attribute for each tuple.
  + **Foreign Key**: References primary keys in other tables to establish relationships.
* **Significance**: Keys maintain relational integrity, enabling accurate data linkage across tables.

**5. Constraints**

* **Definition**: Constraints are rules enforced on data to ensure validity and accuracy.
* **Purpose**: Prevents invalid data entry by setting conditions for attributes.
* **Main Points**:
  + **Entity Integrity**: Ensures primary key attributes cannot be null.
  + **Referential Integrity**: Ensures foreign keys match primary keys in referenced tables.
* **Significance**: Constraints are critical for maintaining consistent and accurate data across relational databases.

**6. Enterprise Constraints**

* **Definition**: Business rules that apply to data in multiple tables, beyond basic referential constraints.
* **Purpose**: Enforces higher-level data restrictions based on organizational policies.
* **Main Points**:
  + Examples include limits on transaction amounts or restricting students to a maximum number of courses.
* **Significance**: Ensures data complies with real-world business rules, essential for operational integrity.

**7. Features of Good Relational Designs**

* **Definition**: Characteristics that make relational designs efficient, maintainable, and error-free.
* **Purpose**: Minimizes redundancy and ensures a design that supports easy data retrieval and maintenance.
* **Main Points**:
  + Avoids repetition and supports meaningful data representation.
  + Allows efficient updates and storage.
* **Significance**: A well-designed relational database is optimized for performance, scalability, and maintainability.

**8. Data Redundancy and Update Anomalies**

* **Definition**: Redundant data leads to duplicate information; update anomalies occur when redundant data is inconsistently updated.
* **Purpose**: Reduces inefficiencies and potential inconsistencies.
* **Main Points**:
  + **Insertion Anomaly**: Issues when inserting new data.
  + **Deletion Anomaly**: Issues when deleting data, leading to loss of essential information.
* **Significance**: Addressing redundancy and anomalies is key to database efficiency and reliability.

**9. Normalization**

* **Definition**: A process of organizing data to minimize redundancy and dependency.
* **Purpose**: Structures data to avoid anomalies and ensure each piece of data is logically stored.
* **Main Points**:
  + Divides tables into smaller, related tables and establishes foreign keys.
* **Significance**: Normalization is essential for reducing data redundancy and improving database performance.

**10. Atomic Domains and First Normal Form (1NF)**

* **Definition**: A relation is in 1NF if it has atomic (indivisible) values in each field.
* **Purpose**: Ensures that each table cell contains only one value, setting the foundation for further normalization.
* **Main Points**:
  + Tables with multi-valued attributes are split into multiple rows or columns.
* **Significance**: 1NF is the first step in organizing data into a relational structure.

**11. Decomposition using Functional Dependencies**

* **Definition**: Splitting tables based on functional dependencies to ensure each relation satisfies normalization rules.
* **Purpose**: Eliminates redundancy by ensuring that each table only contains related data.
* **Main Points**:
  + Functional dependencies determine attribute relationships, guiding decomposition.
* **Significance**: Reduces anomalies and simplifies data integrity checks in relational databases.

**12. Second Normal Form (2NF)**

* **Definition**: A relation is in 2NF if it is in 1NF and all non-key attributes are fully functionally dependent on the primary key.
* **Purpose**: Eliminates partial dependencies, where non-key attributes depend only on part of a composite key.
* **Main Points**:
  + Tables are further divided to eliminate partial dependencies.
* **Significance**: 2NF improves data integrity and reduces redundancy in composite key scenarios.

**13. Third Normal Form (3NF)**

* **Definition**: A relation is in 3NF if it is in 2NF and contains no transitive dependencies.
* **Purpose**: Eliminates indirect dependencies, ensuring that non-key attributes depend only on the primary key.
* **Main Points**:
  + Removes transitive dependencies to simplify data relationships.
* **Significance**: 3NF reduces data redundancy and enhances relational model integrity.

**14. Boyce-Codd Normal Form (BCNF)**

* **Definition**: A stricter version of 3NF where every determinant is a super key.
* **Purpose**: Ensures that no non-trivial functional dependency exists where a non-superkey determines another attribute.
* **Main Points**:
  + BCNF eliminates certain anomalies not addressed by 3NF.
* **Significance**: BCNF improves design quality, particularly in complex databases with multiple candidate keys.

**15. Lossless Join and Dependency Preservation**

* **Definition**: Conditions in decomposition ensuring no data loss and that dependencies are preserved.
* **Purpose**: Maintains data integrity by ensuring all original data can be reconstructed after decomposition.
* **Main Points**:
  + Lossless join guarantees that decomposed tables can be rejoined to obtain the original table without loss.
  + Dependency preservation ensures all functional dependencies remain enforceable.
* **Significance**: Essential for maintaining database integrity and relational completeness after normalization.

**Unit IV: Database Transaction Management**

**1. Database Transaction**

* **Definition**: A transaction is a set of operations that performs a single logical unit of work within a database.
* **Purpose**: Ensures that related operations either all complete successfully or none do, maintaining consistency.
* **Main Points**:
  + A transaction typically includes actions like reading and writing data items.
  + Transactions have states: active, partially committed, failed, aborted, and committed.
* **Significance**: Transactions are foundational to database reliability, enabling secure and consistent data handling.

**2. ACID Properties**

* **Definition**: ACID (Atomicity, Consistency, Isolation, Durability) properties define the standards for transaction reliability.
* **Purpose**: Ensures each transaction’s effect is predictable and recoverable.
* **Main Points**:
  + **Atomicity**: Transactions are all-or-nothing.
  + **Consistency**: Transitions the database from one valid state to another.
  + **Isolation**: Concurrent transactions do not interfere with each other.
  + **Durability**: Once committed, results are permanent, even after a failure.
* **Significance**: ACID properties ensure reliable and robust database operations, critical for systems requiring high data integrity.

**3. Schedule and Serializability**

* **Definition**: A schedule is an ordered sequence of transaction operations. Serializability ensures that concurrent transactions yield the same result as sequential execution.
* **Purpose**: Enables concurrency without compromising the accuracy of transaction outcomes.
* **Main Points**:
  + **Conflict Serializability**: Transactions conflict if their order affects the final outcome.
  + **View Serializability**: Transactions are serializable if they produce the same final view of data.
* **Significance**: Serializability is essential for maintaining data accuracy in concurrent transaction environments.

**4. Recoverable and Cascadeless Schedules**

* **Definition**: Schedules where dependent transactions commit only after the transactions they depend on are recoverable; cascadeless schedules prevent cascading aborts.
* **Purpose**: Minimizes errors in dependent transactions and ensures a stable transaction sequence.
* **Main Points**:
  + **Recoverable Schedules**: Commit operations follow dependencies.
  + **Cascadeless Schedules**: Prevent read operations until preceding transactions commit.
* **Significance**: Ensures system reliability by avoiding unnecessary rollbacks and maintaining data consistency.

**5. Concurrency Control**

* **Definition**: Methods for managing simultaneous transactions without conflicts.
* **Purpose**: Ensures that concurrent transactions do not negatively affect each other’s results.
* **Main Points**:
  + **Lock-based Protocols**: Uses locks (shared or exclusive) to control access.
  + **Timestamp Protocols**: Assigns timestamps to ensure transactions execute in order.
* **Significance**: Concurrency control is vital for maximizing database performance while ensuring accurate results.

**6. Deadlock and Deadlock Handling**

* **Definition**: Deadlock occurs when transactions wait indefinitely due to circular dependencies on resources.
* **Purpose**: Identifies and resolves deadlocks to maintain transaction flow.
* **Main Points**:
  + **Detection**: Identifies circular waits using wait-for graphs.
  + **Prevention**: Techniques like two-phase locking and ordering prevent deadlocks.
* **Significance**: Deadlock handling is critical in environments with high concurrency to avoid system stalls.

**7. Recovery Concepts and Methods**

* **Definition**: Techniques to restore database consistency after a failure.
* **Purpose**: Ensures data integrity by recovering from crashes, errors, or system failures.
* **Main Points**:
  + **Log-based Recovery**: Tracks changes for possible reversion or re-execution.
  + **Checkpointing**: Creates recovery points to simplify restoration.
  + **Shadow Paging**: Maintains a shadow copy of data pages for quick recovery.
* **Significance**: Recovery ensures durability and consistency, especially critical for critical databases where data integrity is paramount.

**Unit V: NoSQL Databases**

**1. Introduction to Distributed Database Systems**

* **Definition**: A distributed database system is a collection of interconnected databases spread across multiple locations.
* **Purpose**: Provides improved data access, reliability, and scalability by allowing data storage across multiple servers.
* **Main Points**:
  + Offers local autonomy, data replication, and high availability.
  + Examples include geographically distributed databases used by large organizations.
* **Significance**: Distributed systems increase efficiency and scalability, enabling businesses to manage data effectively across diverse locations.

**2. CAP Theorem**

* **Definition**: CAP (Consistency, Availability, Partition tolerance) Theorem explains the trade-offs in distributed databases.
* **Purpose**: Defines that it’s impossible to achieve all three properties (consistency, availability, and partition tolerance) simultaneously.
* **Main Points**:
  + **Consistency**: Every read receives the latest data or an error.
  + **Availability**: Every request gets a response, even without the most recent data.
  + **Partition Tolerance**: System continues to operate despite network partitions.
* **Significance**: CAP theorem guides design choices for NoSQL systems, prioritizing two properties based on system needs.

**3. Types of Data**

* **Definition**: Data types include structured, unstructured, and semi-structured data, each with unique characteristics.
* **Purpose**: Determines how data is organized and accessed.
* **Main Points**:
  + **Structured Data**: Organized in schemas; accessible with SQL.
  + **Unstructured Data**: Lacks a schema; typically large, like images or emails.
  + **Semi-Structured Data**: Has some organizational tags; includes XML or JSON.
* **Significance**: Helps in choosing the right NoSQL model for various data formats.

**4. NoSQL Database**

* **Definition**: Non-relational databases designed for scalable, flexible data storage.
* **Purpose**: Addresses the limitations of RDBMS in handling large-scale, unstructured data.
* **Main Points**:
  + Schema-less, supports horizontal scaling, and works with distributed architectures.
  + Ideal for big data, unstructured data, and applications requiring high availability.
* **Significance**: NoSQL databases enable efficient data management for modern applications requiring flexibility and scalability.

**5. Types of NoSQL Databases**

* **Definition**: NoSQL databases are classified into key-value stores, document stores, graph databases, and wide column stores.
* **Purpose**: Provides specialized storage solutions based on data structure and usage needs.
* **Main Points**:
  + **Key-Value Store**: Simple data structure for large-scale storage; used in caching.
  + **Document Store**: Stores semi-structured data like JSON; supports flexible querying.
  + **Graph Database**: Models relationships; ideal for social networks.
  + **Wide Column Store**: Columns stored independently; suited for analytical applications.
* **Significance**: These types cater to specific data storage and retrieval needs, offering optimized performance for various applications.

**6. BASE Properties**

* **Definition**: BASE (Basically Available, Soft state, Eventual consistency) describes the characteristics of NoSQL systems, in contrast to ACID properties in relational databases.
* **Purpose**: Enables scalability and flexibility by prioritizing availability and eventual consistency.
* **Main Points**:
  + **Basically Available**: Guarantees availability even in failures.
  + **Soft State**: State of the database may change over time without new inputs.
  + **Eventual Consistency**: Ensures that data will become consistent over time.
* **Significance**: BASE properties support high availability and scalability, essential for distributed NoSQL systems.

**7. ACID vs. BASE**

* **Definition**: ACID (Atomicity, Consistency, Isolation, Durability) and BASE properties define transaction handling in RDBMS and NoSQL.
* **Purpose**: Highlights trade-offs in database consistency and availability.
* **Main Points**:
  + ACID ensures strict data integrity and is used in RDBMS.
  + BASE prioritizes flexibility and availability for distributed systems.
* **Significance**: This comparison helps in selecting the appropriate database system depending on the need for consistency or scalability.

**8. Comparative Study of RDBMS and NoSQL**

* **Definition**: Contrasts traditional RDBMS with NoSQL systems.
* **Purpose**: Highlights the differences in data model, scalability, and flexibility.
* **Main Points**:
  + RDBMS is relational and uses SQL; NoSQL supports unstructured data.
  + RDBMS emphasizes ACID; NoSQL uses BASE.
* **Significance**: Guides database selection based on system requirements, particularly for big data and distributed systems.

**9. MongoDB**

* **Definition**: A popular document-based NoSQL database known for high performance and flexibility.
* **Purpose**: Enables fast, scalable data handling with minimal schema constraints.
* **Main Points**:
  + Supports JSON-like storage, CRUD operations, indexing, aggregation, and replication.
  + Horizontal scalability via sharding, replication for fault tolerance.
* **Significance**: MongoDB is widely used in web applications, data analytics, and IoT, supporting flexible and scalable data management.

**Unit VI: Advances in Databases**

**1. Emerging Databases**

* **Definition**: Emerging databases are advanced databases developed to meet the demands of evolving software applications and increased data complexity.
* **Purpose**: Designed to handle large, complex datasets efficiently, supported by modern database tools.
* **Main Points**:
  + **Active Databases**: Feature automated rules (triggers) for action in response to events.
  + **Deductive Databases**: Apply logical programming (e.g., Prolog) for rule-based deductions.
  + **Main Memory Databases**: Store data primarily in memory for fast access.
  + **Semantic Databases**: Use structured data models that store meaning-based relationships.
* **Significance**: These databases provide flexibility, speed, and automated processing, suitable for diverse applications, from real-time analytics to AI.

**2. Complex Data Types**

* **Definition**: Complex data types go beyond traditional relational data structures to handle semi-structured and unstructured data.
* **Purpose**: Manages data that does not fit well into tables by allowing hierarchical and flexible data organization.
* **Main Points**:
  + **Semi-Structured Data**: Contains some organizational tags (e.g., XML) but lacks rigid structure.
  + Enables storage and manipulation of diverse data forms like emails, web pages.
* **Significance**: Complex data types expand the types of data that can be stored and queried, especially important in data-intensive fields.

**3. Nested Data Types**

* **Definition**: Nested data types allow hierarchical, array, and map structures within data records.
* **Purpose**: Supports complex data hierarchies in databases, suitable for JSON and XML data formats.
* **Main Points**:
  + **JSON (JavaScript Object Notation)**: Stores data as key-value pairs, widely used for web data.
  + **XML (eXtensible Markup Language)**: Standard for representing structured information.
* **Significance**: Facilitates modern web and app data interchange, enhancing the database’s flexibility to store detailed data structures.

**4. Object Orientation in Databases**

* **Definition**: Object-oriented databases (OODBMS) integrate object-oriented programming principles into database management.
* **Purpose**: Offers data storage in object form, supporting object features like inheritance and encapsulation.
* **Main Points**:
  + **Object-Relational Mapping (ORM)**: Bridges object-oriented languages with relational databases.
  + **Table Inheritance**: Allows sub-tables to inherit attributes from parent tables.
* **Significance**: Object orientation improves data handling in applications requiring complex data structures, making it easier to integrate with object-oriented programming.

**5. Spatial Data**

* **Definition**: Spatial data is related to geographic or geometric space, represented through points, lines, and polygons.
* **Purpose**: Manages spatial information for applications like GIS and mapping.
* **Main Points**:
  + **Geometric Data**: Manages shapes, sizes, and relative positions of objects.
  + **Geographic Data**: Involves data associated with Earth coordinates (e.g., GPS).
* **Significance**: Spatial data support is critical for applications in urban planning, logistics, and environmental science

**1. Employee Table (Basic SQL Operations)**

**1.1 Create the employee table**

**sql**

**Copy code**

**CREATE TABLE employee (**

**emp\_id INT PRIMARY KEY,**

**e\_name VARCHAR(50),**

**salary DECIMAL(10, 2),**

**Date\_of\_Joining DATE,**

**Dapt\_no INT,**

**Designation VARCHAR(50)**

**);**

**1.2 Insert 10 records into the table**

**sql**

**Copy code**

**INSERT INTO employee (emp\_id, e\_name, salary, Date\_of\_Joining, Dapt\_no, Designation)**

**VALUES**

**(1, 'Alice', 50000, '2022-01-01', 1, 'Manager'),**

**(2, 'Bob', 45000, '2021-05-15', 2, 'Developer'),**

**-- Add 8 more records with unique emp\_id and values.**

**1.3 Create a view emp\_v1 with specific columns**

**sql**

**Copy code**

**CREATE VIEW emp\_v1 AS**

**SELECT emp\_id, e\_name, Dapt\_no**

**FROM employee;**

**1.4 Create another view (if different)**

**sql**

**Copy code**

**CREATE VIEW emp\_view AS**

**SELECT \* FROM employee;**

**1.5 Update Dapt\_no in view and check if it reflects in the base table**

**sql**

**Copy code**

**UPDATE emp\_v1**

**SET Dapt\_no = 3**

**WHERE emp\_id = 1;**

***Check the employee table to see if the Dapt\_no is updated.***

**1.6 Create emp\_id as primary key and show indices on table**

* **The emp\_id is already a primary key.**
* **Show indices using:**

**sql**

**Copy code**

**SHOW INDEX FROM employee;**

**1.7 Create a user-defined index on Designation column**

**sql**

**Copy code**

**CREATE INDEX idx\_designation ON employee (Designation);**

**2. Employee Table (Queries)**

**2.1 Display employees whose name contains the letter ‘e’**

**sql**

**Copy code**

**SELECT \* FROM employee**

**WHERE e\_name LIKE '%e%';**

**2.2 Display different types of Designation**

**sql**

**Copy code**

**SELECT DISTINCT Designation**

**FROM employee;**

**2.3 Display e\_name and salary of employees where location is ‘Mumbai’**

**Assuming location is a new column you added:**

**sql**

**Copy code**

**SELECT e\_name, salary**

**FROM employee**

**WHERE location = 'Mumbai';**

**2.4 Display e\_name and Dapt\_no of employees in ‘Manager’ or ‘Marketing’ departments**

**sql**

**Copy code**

**SELECT e\_name, Dapt\_no**

**FROM employee**

**WHERE Designation IN ('Manager', 'Marketing');**

**2.5 Display department name with more than one employee**

**sql**

**Copy code**

**SELECT Dapt\_no, COUNT(\*)**

**FROM employee**

**GROUP BY Dapt\_no**

**HAVING COUNT(\*) > 1;**

**2.6 Rename employee table to emp1**

**sql**

**Copy code**

**ALTER TABLE employee RENAME TO emp1;**

**2.7 Add a new column city to employee table**

**sql**

**Copy code**

**ALTER TABLE emp1 ADD city VARCHAR(50);**

**3. Employee Table (Advanced Queries)**

**3.1 Find department with maximum employees**

**sql**

**Copy code**

**SELECT Dapt\_no**

**FROM employee**

**GROUP BY Dapt\_no**

**ORDER BY COUNT(\*) DESC**

**LIMIT 1;**

**3.2 Display employees whose name starts with ‘A’ or ‘P’**

**sql**

**Copy code**

**SELECT e\_name, Designation, Dapt\_no**

**FROM employee**

**WHERE e\_name LIKE 'A%' OR e\_name LIKE 'P%';**

**3.3 Display max salary from department 2 and min salary from department 4**

**sql**

**Copy code**

**SELECT MAX(salary) AS max\_salary**

**FROM employee**

**WHERE Dapt\_no = 2;**

**SELECT MIN(salary) AS min\_salary**

**FROM employee**

**WHERE Dapt\_no = 4;**

**3.4 Display employees with salary less than the average in department 3**

**sql**

**Copy code**

**SELECT \***

**FROM employee**

**WHERE salary < (**

**SELECT AVG(salary)**

**FROM employee**

**WHERE Dapt\_no = 3**

**);**

**3.5 Display employees hired earliest or latest**

**sql**

**Copy code**

**SELECT \***

**FROM employee**

**ORDER BY Date\_of\_Joining ASC**

**LIMIT 1;**

**SELECT \***

**FROM employee**

**ORDER BY Date\_of\_Joining DESC**

**LIMIT 1;**

**3.6 Display employees who are Manager or Market Analyst**

**sql**

**Copy code**

**SELECT e\_name, Dapt\_no**

**FROM employee**

**WHERE Designation IN ('Manager', 'Market Analyst');**

**3.7 List employees hired in August**

**sql**

**Copy code**

**SELECT \***

**FROM employee**

**WHERE MONTH(Date\_of\_Joining) = 8;**

**3.8 List employees hired after ‘31/12/2006’**

**sql**

**Copy code**

**SELECT \***

**FROM employee**

**WHERE Date\_of\_Joining > '2006-12-31';**

**3.9 Find average annual salary per department**

**sql**

**Copy code**

**SELECT Dapt\_no, AVG(salary) \* 12 AS avg\_annual\_salary**

**FROM employee**

**GROUP BY Dapt\_no;**

**4. Customer and Order Tables**

**4.1 Create Customer and Order tables with primary and foreign key constraints**

**sql**

**Copy code**

**CREATE TABLE Customer (**

**c\_id INT PRIMARY KEY,**

**c\_name VARCHAR(50),**

**email VARCHAR(100),**

**city VARCHAR(50),**

**pincode INT**

**);**

**CREATE TABLE Order (**

**order\_id INT PRIMARY KEY,**

**date DATE,**

**amount DECIMAL(10, 2),**

**cust\_id INT,**

**FOREIGN KEY (cust\_id) REFERENCES Customer(c\_id)**

**);**

**4.2 Insert 10 records in each table**

**sql**

**Copy code**

**INSERT INTO Customer (c\_id, c\_name, email, city, pincode)**

**VALUES**

**(1, 'John', 'john@example.com', 'Mumbai', 400001),**

**-- Add 9 more records**

**;**

**INSERT INTO Order (order\_id, date, amount, cust\_id)**

**VALUES**

**(101, '2023-01-01', 250.00, 1),**

**-- Add 9 more records**

**;**

**4.3 Find all orders placed by customers with cust\_id = 2**

**sql**

**Copy code**

**SELECT \***

**FROM Order**

**WHERE cust\_id = 2;**

**4.4 Find list of customers who placed their order and details of order**

**sql**

**Copy code**

**SELECT Customer.\*, Order.\***

**FROM Customer**

**JOIN Order ON Customer.c\_id = Order.cust\_id;**

**4.5 List of customers who haven’t placed an order**

**sql**

**Copy code**

**SELECT \***

**FROM Customer**

**WHERE c\_id NOT IN (SELECT cust\_id FROM Order);**

**4.6 List all orders and append to customer table**

**sql**

**Copy code**

**SELECT Customer.\*, Order.\***

**FROM Customer**

**LEFT JOIN Order ON Customer.c\_id = Order.cust\_id;**

**4.7 Display all records**

**sql**

**Copy code**

**SELECT \* FROM Customer;**

**SELECT \* FROM Order;**

**4.8 Display customers from the same city**

**sql**

**Copy code**

**SELECT c\_name, city**

**FROM Customer**

**GROUP BY city**

**HAVING COUNT(city) > 1;**

**5. Borrower and Fine Tables**

**5.1 Create Borrower and Fine tables**

**sql**

**Copy code**

**CREATE TABLE Borrower (**

**RollNo INT PRIMARY KEY,**

**Name VARCHAR(50),**

**DateofIssue DATE,**

**NameofBook VARCHAR(50),**

**Status VARCHAR(10) CHECK (Status IN ('Issued', 'Returned'))**

**);**

**CREATE TABLE Fine (**

**Roll\_no INT,**

**Date DATE,**

**Amt DECIMAL(10, 2),**

**PRIMARY KEY (Roll\_no, Date)**

**);**

**5.2 Insert 10 records in each table**

**sql**

**Copy code**

**INSERT INTO Borrower (RollNo, Name, DateofIssue, NameofBook, Status)**

**VALUES**

**(1, 'Alice', '2023-07-01', 'DBMS Basics', 'Issued'),**

**-- Add 9 more records**

**;**

**INSERT INTO Fine (Roll\_no, Date, Amt)**

**VALUES**

**(1, '2023-08-01', 100.00),**

**-- Add 9 more records**

**;**

**5.3 Find count of books with Issued status**

**sql**

**Copy code**

**SELECT COUNT(\*) AS IssuedBooks**

**FROM Borrower**

**WHERE Status = 'Issued';**

**5.4 Display all records**

**sql**

**Copy code**

**SELECT \* FROM Borrower;**

**SELECT \* FROM Fine;**

**5.5 Display RollNo with the same DateofIssue**

**sql**

**Copy code**

**SELECT RollNo**

**FROM Borrower**

**GROUP BY DateofIssue**

**HAVING COUNT(\*) > 1;**

**6. Student Table (DDL and DML Operations)**

**Consider the table student (roll\_no, name, marks, class) with roll\_no as the primary key.**

**DDL Operations (Data Definition Language)**

1. **Create the student table**

**sql**

**Copy code**

**CREATE TABLE student (**

**roll\_no INT PRIMARY KEY,**

**name VARCHAR(50),**

**marks DECIMAL(5, 2),**

**class VARCHAR(10)**

**);**

1. **Add a new column age to the student table**

**sql**

**Copy code**

**ALTER TABLE student**

**ADD age INT;**

1. **Drop the age column from the student table**

**sql**

**Copy code**

**ALTER TABLE student**

**DROP COLUMN age;**

**DML Operations (Data Manipulation Language)**

1. **Insert a record into the student table**

**sql**

**Copy code**

**INSERT INTO student (roll\_no, name, marks, class)**

**VALUES (1, 'John Doe', 85.5, '10th');**

1. **Update marks of a student**

**sql**

**Copy code**

**UPDATE student**

**SET marks = 90**

**WHERE roll\_no = 1;**

1. **Delete a student record**

**sql**

**Copy code**

**DELETE FROM student**

**WHERE roll\_no = 1;**

**7. Create job\_history Table with Constraints**

**Assume there is an existing jobs table with the schema Job (job\_id, job\_title, min\_sal, max\_sal).**

**Create the job\_history table**

**sql**

**Copy code**

**CREATE TABLE job\_history (**

**employee\_id INT PRIMARY KEY,**

**start\_date DATE,**

**end\_date DATE,**

**job\_id INT,**

**department\_id INT,**

**CONSTRAINT fk\_job\_id**

**FOREIGN KEY (job\_id) REFERENCES jobs(job\_id)**

**);**

**This ensures that employee\_id values are unique, and job\_id values in job\_history must match values in the jobs table.**

**8. Queries on Employee, Works, Company, and Manages Tables**

**Given the schemas:**

* **employee (employee\_name, street, city)**
* **works (employee\_name, company\_name, salary)**
* **company (company\_name, city)**
* **manages (employee\_name, manager\_name)**

**a) Employees working for the same company and earning more than $10,000**

**sql**

**Copy code**

**SELECT e.employee\_name, e.street, e.city**

**FROM employee e**

**JOIN works w ON e.employee\_name = w.employee\_name**

**WHERE w.salary > 10000;**

**b) Employees living in the same cities as their companies**

**sql**

**Copy code**

**SELECT e.employee\_name**

**FROM employee e**

**JOIN works w ON e.employee\_name = w.employee\_name**

**JOIN company c ON w.company\_name = c.company\_name**

**WHERE e.city = c.city;**

**c) Employees earning more than the average salary of their company**

**sql**

**Copy code**

**SELECT w.employee\_name**

**FROM works w**

**JOIN (**

**SELECT company\_name, AVG(salary) AS avg\_salary**

**FROM works**

**GROUP BY company\_name**

**) AS company\_avg ON w.company\_name = company\_avg.company\_name**

**WHERE w.salary > company\_avg.avg\_salary;**

**9. Additional Queries on Employee, Works, Company, and Manages Tables**

**a) Name of the company with the smallest payroll**

**sql**

**Copy code**

**SELECT company\_name**

**FROM works**

**GROUP BY company\_name**

**ORDER BY SUM(salary) ASC**

**LIMIT 1;**

**b) Employees living on the same streets and cities as their managers**

**sql**

**Copy code**

**SELECT e.employee\_name**

**FROM employee e**

**JOIN manages m ON e.employee\_name = m.employee\_name**

**JOIN employee mgr ON m.manager\_name = mgr.employee\_name**

**WHERE e.city = mgr.city AND e.street = mgr.street;**

**10. CRUD Operations and Map-Reduce (MongoDB)**

**Consider a MongoDB collection named Item with fields ItemID, ItemQuantity, Price, and Brand.**

**CRUD Operations in MongoDB**

1. **Insert a document (CREATE)**

**javascript**

**Copy code**

**db.Item.insertOne({**

**ItemID: 1,**

**ItemQuantity: 10,**

**Price: 500,**

**Brand: "BrandA"**

**});**

1. **Retrieve a document (READ)**

**javascript**

**Copy code**

**db.Item.find({ ItemID: 1 });**

1. **Update a document (UPDATE)**

**javascript**

**Copy code**

**db.Item.updateOne(**

**{ ItemID: 1 },**

**{ $set: { Price: 550 } }**

**);**

1. **Delete a document (DELETE)**

**javascript**

**Copy code**

**db.Item.deleteOne({ ItemID: 1 });**

**MapReduce to Count Quantity of Each Item**

**javascript**

**Copy code**

**db.Item.mapReduce(**

**function() { emit(this.ItemID, this.ItemQuantity); },**

**function(key, values) { return Array.sum(values); },**

**{ out: "ItemQuantityCounts" }**

**);**

**This mapReduce operation will generate a collection ItemQuantityCounts with the total quantity of each item.**

**11. CRUD Operations (SAVE Method in MongoDB)**

**For the Item collection, the save method can either insert a new document or update an existing one based on \_id.**

**Using the save Method**

1. **Insert a new document if ItemID does not exist**

**javascript**

**Copy code**

**db.Item.save({**

**\_id: ObjectId("unique\_id\_here"),**

**ItemID: 2,**

**ItemQuantity: 20,**

**Price: 700,**

**Brand: "BrandB"**

**});**

1. **Update an existing document if ItemID exists**

**If a document with \_id is found, the save method updates it; if not, it inserts a new document.**

1. **MapReduce to Count Quantity of Each Item**

**This was covered in question 10. The mapReduce operation would aggregate quantities for each ItemID as shown above.**

**12. MongoDB CRUD Operations (SAVE) and Aggregation on Item Collection**

**For the collection Item with fields ItemID, ItemQuantity, Price, Brand, and Discount:**

**1. Display the count of items brand-wise**

**javascript**

**Copy code**

**db.Item.aggregate([**

**{ $group: { \_id: "$Brand", count: { $sum: 1 } } }**

**]);**

**2. Display item with minimum price**

**javascript**

**Copy code**

**db.Item.find().sort({ Price: 1 }).limit(1);**

**3. Display maximum discount given for an item**

**javascript**

**Copy code**

**db.Item.find().sort({ Discount: -1 }).limit(1);**

**13. Map-Reduce Operation for Counting Marks of Students**

**Consider the collection student (roll\_no, name, marks, class).**

**Map-Reduce Operation**

**javascript**

**Copy code**

**db.student.mapReduce(**

**function() { emit(this.roll\_no, this.marks); },**

**function(key, values) { return Array.sum(values); },**

**{ out: "total\_marks\_per\_student" }**

**);**

**This will create a collection total\_marks\_per\_student with each student’s roll\_no and their total marks.**

**14. Map-Reduce Operation for Displaying Persons with Same Profession**

**Consider the collection person (person\_id, name, addr, profession).**

**Map-Reduce Operation**

**javascript**

**Copy code**

**db.person.mapReduce(**

**function() { emit(this.profession, this.name); },**

**function(key, values) { return values; },**

**{ out: "people\_with\_same\_profession" }**

**);**

**This will create a collection people\_with\_same\_profession, where each document lists all names grouped by profession.**

**15. MongoDB CRUD Operations on Person Collection**

**Consider the collection person (person\_id, name, addr, profession).**

**1. Create Collection**

**javascript**

**Copy code**

**db.createCollection("person");**

**2. Insert Data**

**javascript**

**Copy code**

**db.person.insertOne({**

**person\_id: 1,**

**name: "John Doe",**

**addr: "123 Elm Street",**

**profession: "Engineer"**

**});**

**3. Read Data**

**javascript**

**Copy code**

**db.person.find();**

**4. Update Data**

**javascript**

**Copy code**

**db.person.updateOne(**

**{ person\_id: 1 },**

**{ $set: { profession: "Senior Engineer" } }**

**);**

**5. Delete Data**

**javascript**

**Copy code**

**db.person.deleteOne({ person\_id: 1 });**

**16. MongoDB CRUD Operations and Aggregation on Employee Collection**

**Consider the collection employee (emp\_id, e\_name, salary, Date\_of\_Joining, Dapt\_no, Designation).**

**1. Display the count of employees department-wise**

**javascript**

**Copy code**

**db.employee.aggregate([**

**{ $group: { \_id: "$Dapt\_no", count: { $sum: 1 } } }**

**]);**

**2. Display the average salary of employees in the Sales department**

**javascript**

**Copy code**

**db.employee.aggregate([**

**{ $match: { Dapt\_no: "Sales" } },**

**{ $group: { \_id: null, averageSalary: { $avg: "$salary" } } }**

**]);**

**3. Display minimum salary of employees who joined in June 2016**

**javascript**

**Copy code**

**db.employee.aggregate([**

**{ $match: { Date\_of\_Joining: { $regex: "^2016-06" } } },**

**{ $group: { \_id: null, minSalary: { $min: "$salary" } } }**

**]);**

**4. Display maximum salary given to employees in the Production department**

**javascript**

**Copy code**

**db.employee.aggregate([**

**{ $match: { Dapt\_no: "Production" } },**

**{ $group: { \_id: null, maxSalary: { $max: "$salary" } } }**

**]);**

**5. Display first and last employee record department-wise**

**javascript**

**Copy code**

**db.employee.aggregate([**

**{ $sort: { Date\_of\_Joining: 1 } },**

**{ $group: {**

**\_id: "$Dapt\_no",**

**firstEmployee: { $first: "$$ROOT" },**

**lastEmployee: { $last: "$$ROOT" }**

**}}**

**]);**

**17. Java Program for CRUD Operations on Student Table**

**Consider the student table with fields roll\_no, name, marks, class. Here's a sample Java menu-driven program for CRUD operations.**

**java**

**Copy code**

**import java.sql.\*;**

**import java.util.Scanner;**

**public class StudentCRUD {**

**public static void main(String[] args) {**

**try (Connection conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/yourDatabase", "username", "password");**

**Scanner scanner = new Scanner(System.in)) {**

**System.out.println("Choose operation: 1) Add 2) Update 3) Delete");**

**int choice = scanner.nextInt();**

**switch (choice) {**

**case 1:**

**System.out.println("Enter roll\_no, name, marks, class:");**

**int roll\_no = scanner.nextInt();**

**String name = scanner.next();**

**double marks = scanner.nextDouble();**

**String studentClass = scanner.next();**

**PreparedStatement insertStmt = conn.prepareStatement("INSERT INTO student (roll\_no, name, marks, class) VALUES (?, ?, ?, ?)");**

**insertStmt.setInt(1, roll\_no);**

**insertStmt.setString(2, name);**

**insertStmt.setDouble(3, marks);**

**insertStmt.setString(4, studentClass);**

**insertStmt.executeUpdate();**

**System.out.println("Record inserted successfully.");**

**break;**

**case 2:**

**System.out.println("Enter roll\_no and new marks:");**

**roll\_no = scanner.nextInt();**

**marks = scanner.nextDouble();**

**PreparedStatement updateStmt = conn.prepareStatement("UPDATE student SET marks = ? WHERE roll\_no = ?");**

**updateStmt.setDouble(1, marks);**

**updateStmt.setInt(2, roll\_no);**

**updateStmt.executeUpdate();**

**System.out.println("Record updated successfully.");**

**break;**

**case 3:**

**System.out.println("Enter roll\_no to delete:");**

**roll\_no = scanner.nextInt();**

**PreparedStatement deleteStmt = conn.prepareStatement("DELETE FROM student WHERE roll\_no = ?");**

**deleteStmt.setInt(1, roll\_no);**

**deleteStmt.executeUpdate();**

**System.out.println("Record deleted successfully.");**

**break;**

**}**

**} catch (SQLException e) {**

**e.printStackTrace();**

**}**

**}**

**}**

**This program connects to a MySQL database and provides options to add, update, or delete student records.**

**18. PL/SQL Stored Procedure for Grade Categorization of Students**

**Consider the procedure proc\_Grade to categorize students based on their marks.**

**Create Procedure**

**sql**

**Copy code**

**CREATE OR REPLACE PROCEDURE proc\_Grade (p\_roll\_no IN NUMBER, p\_marks IN NUMBER, p\_category OUT VARCHAR2) AS**

**BEGIN**

**IF p\_marks >= 990 AND p\_marks <= 1500 THEN**

**p\_category := 'Distinction';**

**ELSIF p\_marks BETWEEN 900 AND 989 THEN**

**p\_category := 'First Class';**

**ELSIF p\_marks BETWEEN 825 AND 899 THEN**

**p\_category := 'Higher Second Class';**

**ELSE**

**p\_category := 'No Category';**

**END IF;**

**END;**

**Usage in PL/SQL Block**

**sql**

**Copy code**

**DECLARE**

**roll\_no NUMBER := 1; -- Example roll number**

**marks NUMBER := 950; -- Example marks**

**category VARCHAR2(20);**

**BEGIN**

**proc\_Grade(roll\_no, marks, category);**

**DBMS\_OUTPUT.PUT\_LINE('Student Category: ' || category);**

**END;**

**This PL/SQL block uses the proc\_Grade procedure to determine the grade of a student based on marks.**

**19. Database Trigger on customer Table for Audit Logging**

This trigger keeps track of records in customer that are updated or deleted. The old values of these records are stored in cust\_Audit.

**SQL Trigger**

sql

Copy code

CREATE OR REPLACE TRIGGER customer\_audit\_trigger

AFTER UPDATE OR DELETE ON customer

FOR EACH ROW

BEGIN

IF DELETING THEN

INSERT INTO cust\_Audit (cust\_id, c\_name, addr, action, action\_date)

VALUES (:OLD.cust\_id, :OLD.c\_name, :OLD.addr, 'DELETE', SYSDATE);

ELSIF UPDATING THEN

INSERT INTO cust\_Audit (cust\_id, c\_name, addr, action, action\_date)

VALUES (:OLD.cust\_id, :OLD.c\_name, :OLD.addr, 'UPDATE', SYSDATE);

END IF;

END;

**20. Database Trigger on client\_master Table for Audit Logging**

This trigger tracks records that are updated or inserted in the client\_master table. The old values of updated records are stored in client\_Audit.

**SQL Trigger**

sql

Copy code

CREATE OR REPLACE TRIGGER client\_master\_audit\_trigger

AFTER INSERT OR UPDATE ON client\_master

FOR EACH ROW

BEGIN

IF INSERTING THEN

INSERT INTO client\_Audit (c\_id, c\_name, acc\_no, action, action\_date)

VALUES (:NEW.c\_id, :NEW.c\_name, :NEW.acc\_no, 'INSERT', SYSDATE);

ELSIF UPDATING THEN

INSERT INTO client\_Audit (c\_id, c\_name, acc\_no, action, action\_date)

VALUES (:OLD.c\_id, :OLD.c\_name, :OLD.acc\_no, 'UPDATE', SYSDATE);

END IF;

END;

**21. PL/SQL Block Using Explicit Cursor for Merging Data in N\_RollCall and O\_RollCall**

This PL/SQL block merges data from N\_RollCall into O\_RollCall, skipping any duplicate records.

sql

Copy code

DECLARE

CURSOR rollcall\_cursor IS

SELECT \* FROM N\_RollCall;

record\_exists BOOLEAN;

BEGIN

FOR rec IN rollcall\_cursor LOOP

SELECT COUNT(\*) INTO record\_exists

FROM O\_RollCall

WHERE roll\_no = rec.roll\_no;

IF record\_exists = 0 THEN

INSERT INTO O\_RollCall (roll\_no, name, class)

VALUES (rec.roll\_no, rec.name, rec.class);

END IF;

END LOOP;

END;

**22. PL/SQL Block for Fine Calculation Based on DateofIssue**

This PL/SQL block accepts roll\_no and name\_of\_book and calculates the fine if the days since the issue are between 15 and 30.

sql

Copy code

DECLARE

v\_roll\_no NUMBER;

v\_name\_of\_book VARCHAR2(50);

v\_date\_of\_issue DATE;

v\_fine\_per\_day NUMBER := 5;

v\_total\_fine NUMBER := 0;

BEGIN

-- Accepting roll number and book name

v\_roll\_no := &roll\_no;

v\_name\_of\_book := '&name\_of\_book';

SELECT DateofIssue INTO v\_date\_of\_issue

FROM Borrower

WHERE RollNo = v\_roll\_no AND NameofBook = v\_name\_of\_book;

IF v\_date\_of\_issue IS NOT NULL THEN

IF SYSDATE - v\_date\_of\_issue BETWEEN 15 AND 30 THEN

v\_total\_fine := (SYSDATE - v\_date\_of\_issue) \* v\_fine\_per\_day;

INSERT INTO Fine (Roll\_no, Date, Amt)

VALUES (v\_roll\_no, SYSDATE, v\_total\_fine);

END IF;

END IF;

END;

**23. Basic SQL Queries on employee Table**

1. **Create Table**

sql

Copy code

CREATE TABLE employee (

emp\_id INT PRIMARY KEY,

e\_name VARCHAR(50),

salary DECIMAL(10,2),

Date\_of\_Joining DATE,

Dapt\_no INT,

Designation VARCHAR(50)

);

1. **Insert Records**

sql

Copy code

INSERT INTO employee VALUES (1, 'John Doe', 50000, '2020-01-01', 1, 'Manager');

-- Insert 9 more records as required.

1. **Create View**

sql

Copy code

CREATE VIEW emp\_vl AS

SELECT emp\_id, e\_name, Dapt\_no

FROM employee;

1. **Display Name and Department of Employees in Manager or Marketing**

sql

Copy code

SELECT e\_name, Dapt\_no

FROM employee

WHERE Designation IN ('Manager', 'Marketing');

1. **Display Employees Hired Earliest or Latest**

sql

Copy code

SELECT \*

FROM employee

ORDER BY Date\_of\_Joining ASC LIMIT 1

UNION ALL

SELECT \*

FROM employee

ORDER BY Date\_of\_Joining DESC LIMIT 1;

1. **List Employees Hired in August**

sql

Copy code

SELECT \*

FROM employee

WHERE EXTRACT(MONTH FROM Date\_of\_Joining) = 8;

1. **List Employees Hired After 31/12/2006**

sql

Copy code

SELECT \*

FROM employee

WHERE Date\_of\_Joining > '2006-12-31';

**24. Indexing and Join on Multiple Tables**

**a. Create Primary Key and Indices on Employee Table**

sql

Copy code

ALTER TABLE employee ADD PRIMARY KEY (emp\_id);

CREATE INDEX idx\_dapt\_no ON employee(Dapt\_no);

**b. Create User-Defined Index on Any Column**

sql

Copy code

CREATE INDEX idx\_salary ON employee(salary);

**c. Create Sequence Using Auto-Increment**

In MySQL, define emp\_id as AUTO\_INCREMENT:

sql

Copy code

CREATE TABLE employee (

emp\_id INT AUTO\_INCREMENT PRIMARY KEY,

e\_name VARCHAR(50),

-- other columns

);

**d. Truncate Table**

sql

Copy code

TRUNCATE TABLE employee;

**e. Find List of Customers Who Placed Orders with Order Details**

sql

Copy code

SELECT c.c\_name, o.\*

FROM Customer c

JOIN Order o ON c.c\_id = o.cust\_id;

**g. List Customers Who Haven’t Placed Orders**

sql

Copy code

SELECT c.c\_name

FROM Customer c

LEFT JOIN Order o ON c.c\_id = o.cust\_id

WHERE o.order\_id IS NULL;

**25. MongoDB Aggregation and Indexing**

**Example Collection: products**

javascript

Copy code

db.products.insertMany([

{ product\_id: 1, name: "Laptop", category: "Electronics", price: 1200 },

{ product\_id: 2, name: "Chair", category: "Furniture", price: 150 },

{ product\_id: 3, name: "Tablet", category: "Electronics", price: 300 }

]);

**a. Count Products in Each Category**

javascript

Copy code

db.products.aggregate([

{ $group: { \_id: "$category", count: { $sum: 1 } } }

]);

**b. Create Index on Product Price**

javascript

Copy code

db.products.createIndex({ price: 1 });