Session 1: Introduction to DBMS & RDBMS

This session introduces the fundamental concepts of Database Management Systems (DBMS), their applications, the relational model (RDBMS), Codd's rules, and the initial need for data normalization.

- **Introduction to DBMS**
    - **Concept:** A Database Management System (DBMS) is a software system designed to store, manage, and retrieve data from a database. It acts as an interface between the database and its end-users or application programs, ensuring data consistency, integrity, and security.
    - **Why DBMS? (vs. File Systems):**
        - **Data Redundancy:** File systems often duplicate data, leading to wasted space and inconsistencies. DBMS aims to minimize this.
        - **Data Inconsistency:** Redundancy often leads to different versions of the same data. DBMS enforces consistency.
        - **Data Sharing:** Multiple users can access the same data concurrently without conflicts.
        - **Data Security:** Provides mechanisms for authentication, authorization, and data encryption.
        - **Data Integrity:** Enforces rules and constraints to ensure data accuracy and reliability.
        - **Data Independence:** Separation of logical (how users view data) and physical (how data is stored) data.
        - **Concurrency Control:** Manages simultaneous access to the database by multiple users.
        - **Backup and Recovery:** Provides tools to protect data from failures and recover it.
- **Areas where DBMS are used**
    - **Banking:** Customer accounts, transactions, loans.
    - **Airlines:** Reservations, flight schedules, passenger information.
    - **Universities:** Student registrations, course catalogs, grades.
    - **E-commerce:** Product catalogs, customer orders, payment processing.
    - **Manufacturing:** Production management, inventory, supply chain.
    - **Human Resources:** Employee records, payroll, benefits.
    - **Telecommunications:** Call records, network usage, billing.
    - **Healthcare:** Patient records, medical history, appointments.
- **Introduction to RDBMS**
    - **Concept:** A Relational Database Management System (RDBMS) is a type of DBMS that organizes data into tables (relations). Data is stored in rows (records/tuples) and columns (attributes/fields). Relationships between data are established through common columns (keys) rather than pointers.
    - **Key Principles:**
        - **Tables (Relations):** Data organized in two-dimensional structures.
        - **Rows (Tuples/Records):** Individual entries in a table.
        - **Columns (Attributes/Fields):** Define the properties or characteristics of the data.
        - **Keys:**
            - **Primary Key:** Uniquely identifies each row in a table. Cannot contain NULL values.

    - **Foreign Key:** A column (or set of columns) in one table that refers to the primary key in another table, establishing a link or relationship.
  - **Schema:** The logical structure of the database.

- **Codd's 12 rules for a Relational Database (conclusion)**

  - Developed by Edgar F. Codd, these rules define what is required for a database system to be considered truly relational. While no commercial RDBMS perfectly adheres to all 12, they serve as a benchmark and guide for relational design.
  - **Conclusion/Purpose:** These rules emphasize logical data independence, data integrity, comprehensive data manipulation, and non-subversion, ensuring that a system genuinely harnesses the power and benefits of the relational model. They promote strong data consistency, flexibility in accessing data, and protection against unauthorized bypass of the database's integrity rules.
  - *(Note: Full detailed explanation of each rule comes in Session 2 as per the outline.)*

- **Need for Normalization.**

  - **Concept:** Normalization is a systematic process of organizing data in a database to reduce data redundancy and improve data integrity. It involves breaking down large tables into smaller, related tables and defining relationships between them.
  - **Why is it needed? (Addressing Data Anomalies):**
    - **Insertion Anomaly:** Cannot add new data without adding other data. E.g., cannot add a new department without a new employee if department details are tied to employee records.
    - **Deletion Anomaly:** Deleting a record unintentionally deletes other related data. E.g., deleting the last employee in a department also deletes all information about that department.
    - **Update Anomaly:** Updating a piece of data requires updating multiple records, potentially leading to inconsistencies if all copies are not updated. E.g., changing a department's location requires updating every employee record in that department.
  - **Benefits:**
    - Reduced data redundancy.
    - Improved data integrity and consistency.
    - More flexible and efficient database design.
    - Easier maintenance and modifications.
    - Faster data retrieval (though complex joins might sometimes negate this).

Session 2: Database Design and Data Modeling

This session expands on the conceptualization of data, moving from abstract ideas to structured relational models, including the detailed understanding of Codd's rules and diagramming techniques.

- **Data Models**

  - **Concept:** A data model is an abstract model that organizes elements of data and standardizes how they relate to one another and to properties of real-world entities. It provides a blueprint for how data will be structured and accessed.
  - **Conceptual Data Model:**

        - **Purpose:** High-level, technology-agnostic representation of the data requirements. Focuses on entities, their attributes, and relationships between them, as perceived by the business or user.
        - **Output:** Often an Entity-Relationship Diagram (ERD).
        - **Audience:** Business users, data architects.
    - **Logical Data Model:**
        - **Purpose:** Detailed representation of data, designed for a specific data management technology (e.g., relational, NoSQL). Maps the conceptual model into tables, columns, primary keys, foreign keys, and relationships. It defines *what* data is stored and *how* it's organized without specifying physical storage.
        - **Output:** Relational schema (tables, columns, keys, relationships).
        - **Audience:** Database designers, developers.
    - **Physical Data Model:**
        - **Purpose:** The lowest-level data model, describing how data is actually stored on disk. Includes details about specific data types, indexing strategies, partitioning, storage allocation, and other DBMS-specific features.
        - **Output:** Schema DDL (Data Definition Language) scripts.
        - **Audience:** Database administrators, performance tuners.

- **Codd's 12 rules for RDBMS**

    - **Rule 1: The Information Rule:** All information in the database is represented explicitly at the logical level in exactly one way—by values in column positions within rows of tables.
    - **Rule 2: Guaranteed Access Rule:** Every item of data (value) must be logically accessible by resorting to a combination of the table name, primary key value, and column name.
    - **Rule 3: Systematic Treatment of Null Values:** Null values (distinct from empty string, zero, or spaces) are supported to represent missing or inapplicable information in a systematic way.
    - **Rule 4: Active Online Catalog Based on the Relational Model:** The database's description (metadata, schema) must be stored in tables within the database itself, just like regular data, and accessible through the same relational query language.
    - **Rule 5: Comprehensive Data Sublanguage Rule:** The system must support at least one relational language (e.g., SQL) that supports data definition, view definition, data manipulation, integrity constraints, authorization, and transaction boundaries.
    - **Rule 6: View Updating Rule:** All views that are theoretically updatable should be updatable by the system.
    - **Rule 7: High-level Insert, Update, and Delete:** The system must be capable of handling not only a single row at a time but also a set of rows (a relation) during insert, update, and delete operations.
    - **Rule 8: Physical Data Independence:** Changes in physical storage characteristics (e.g., file organization, indexing) should not require changes to application programs.
    - **Rule 9: Logical Data Independence:** Changes at the logical level (e.g., adding/removing columns, splitting/merging tables) should not necessitate changes to existing application programs that don't depend on those specific changes.
    - **Rule 10: Integrity Independence:** Integrity constraints (e.g., primary keys, foreign keys, check constraints) must be definable in the relational data sublanguage and stored in the catalog, not in application programs.

- **Rule 11: Distribution Independence:** The database should be able to be distributed across multiple physical locations without impacting application logic.
- **Rule 12: Non-subversion Rule:** If the system provides a low-level (record-at-a-time) language, that language must not be able to bypass the integrity rules defined at the high level.

- **Database Design**
  - **Process:** A systematic approach to structuring a database.
    1. **Requirements Analysis:** Understand what data needs to be stored and how it will be used.
    2. **Conceptual Design:** Create a high-level model (ERD) from requirements, identifying entities, attributes, and relationships.
    3. **Logical Design:** Translate the conceptual model into a relational schema (tables, columns, keys) and apply normalization.
    4. **Physical Design:** Define storage details (data types, indexes, file organization) based on the chosen DBMS.
    5. **Implementation:** Create the database using DDL.
    6. **Testing & Tuning:** Verify data integrity, test performance, and optimize queries/structure.

- **Entity-Relationship Diagram (ERD)**
  - **Concept:** A graphical representation of the conceptual data model. It depicts real-world entities and the relationships between them.
  - **Components:**
    - **Entities:** Rectangles (e.g., `STUDENT`, `COURSE`). Represents a real-world object or concept.
    - **Attributes:** Ovals connected to entities (e.g., `student_id`, `name`, `DOB` for `STUDENT`). Describe properties of an entity.
      - **Key Attributes:** Underlined attributes (e.g., `student_id` as primary key).
    - **Relationships:** Diamonds connecting entities (e.g., `ENROLLS_IN` between `STUDENT` and `COURSE`). Describes how entities interact.
    - **Cardinality/Multiplicity:** Specifies the number of instances of one entity that can be associated with the number of instances of another entity through a relationship.
      - One-to-One (1:1)
      - One-to-Many (1:N)
      - Many-to-Many (N:M)
    - **Participation:** Whether an entity's participation in a relationship is optional or mandatory.

- **Data Flow Diagrams (DFDs)**
  - **Concept:** A graphical representation that illustrates the flow of information within a system. DFDs are used to model the logical processes and data paths, rather than the physical storage structure.
  - **Components:**
    - **Process:** Circle or rounded rectangle (transforms data).
    - **Data Store:** Two parallel lines or open-ended rectangle (data at rest).
    - **External Entity (Terminator):** Rectangle (source or sink of data outside the system).
    - **Data Flow:** Arrows (movement of data).

  - **Role in Database Design:** DFDs help understand the data requirements and how data is processed *before* the database schema is designed, often informing the entities and relationships identified in an ERD.

Session 3: Normalization Forms

This session dives into the practical application of normalization, detailing the first three normal forms and introducing higher forms, along with the concept of de-normalization.

- **Various Normalization Forms**
  - **1st Normal Form (1NF)**
    - **Definition:** A relation is in 1NF if and only if all its attributes are atomic (indivisible) and there are no repeating groups of columns.
    - **Rules:**
      1. Each column must contain atomic values (single value, not a list or set).
      2. There are no repeating groups (a single row should not have multiple values for the same attribute, requiring multiple columns like `phone1`, `phone2`).
      3. Each row is unique (implies a primary key).
    - **Example (Not 1NF):**

OrderID	Customer	Item	Quantity
1	Alice	Pen, Book	2, 1

      Example (1NF):

OrderID	Customer	Item	Quantity
1	Alice	Pen	2
1	Alice	Book	1

  - **2nd 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. This means no non-key attribute depends on only a *part* of a composite primary key.
    - **Rules:**
      1. Must be in 1NF.
      2. No partial dependencies.
    - **Example (Not 2NF, assuming `(OrderID, Item)` is composite PK):**

OrderID	Item	ItemPrice	CustomerName	CustomerAddress
1	Pen	5	Alice	123 Main St

      `CustomerName` and `CustomerAddress` depend only on `OrderID` (a part of the composite PK), not on `Item`.
    - **Example (2NF): Orders Table:**

OrderID	CustomerName	CustomerAddress
1	Alice	123 Main St

Order_Items Table:

OrderID	Item	ItemPrice
1	Pen	5

- **3rd Normal Form (3NF)**
    - **Definition:** A relation is in 3NF if it is in 2NF and there are no transitive dependencies. A transitive dependency occurs when a non-key attribute is dependent on another non-key attribute.
    - **Rules:**
        1. Must be in 2NF.
        2. No transitive dependencies.
    - **Example (Not 3NF, assuming `OrderID` is PK):**

OrderID	CustomerID	CustomerName	CustomerAddress
1	C1	Alice	123 Main St

`CustomerAddress` depends on `CustomerName`, which in turn depends on `CustomerID` (a non-key attribute). `CustomerID` is the determinant for `CustomerName` and `CustomerAddress`.

- **Example (3NF): Orders Table:**

OrderID	CustomerID
1	C1

Customers Table:

CustomerID	CustomerName	CustomerAddress
C1	Alice	123 Main St

- **Introduction to 4th, BCNF, etc.**
    - **Boyce-Codd Normal Form (BCNF):**
        - **Definition:** A relation is in BCNF if and only if for every non-trivial functional dependency X → Y, X is a superkey (i.e., every determinant is a candidate key).
        - **Difference from 3NF:** BCNF is a stronger form of 3NF. 3NF allows transitive dependencies on candidate keys. BCNF disallows it. A table is in 3NF but not BCNF if it has multiple overlapping candidate keys and one of the non-key attributes depends on a part of a candidate key that is not also a superkey.
    - **4th Normal Form (4NF):**
        - **Definition:** A relation is in 4NF if it is in BCNF and contains no multi-valued dependencies. Multi-valued dependency (MVD) means that a determinant determines a set of values, not just one. (e.g., a student can have multiple hobbies and multiple major subjects, independently).
    - **Higher Normal Forms (5NF, 6NF, etc.):** Less commonly applied in practical database design as they tackle very specific and rare dependency types, often adding complexity without significant practical benefit for most business applications.
- **Need for De-normalization**

  - **Concept:** De-normalization is the process of intentionally introducing redundancy into a database by joining tables or adding redundant columns to improve query performance. It is the opposite of normalization.
  - **Why is it needed? (Performance Optimization):**
    - **Reduced Joins:** Fewer joins means faster query execution, especially for complex reports or frequently accessed data.
    - **Simpler Queries:** Sometimes, a denormalized table can simplify complex queries.
    - **Faster Reads:** Often used in data warehousing and OLAP (Online Analytical Processing) systems where read performance is paramount.
    - **Pre-computed Aggregates:** Storing sums, counts, or averages directly in a table to avoid re-calculating them.
  - **Drawbacks:**
    - Introduces data redundancy.
    - Increases the risk of update, insert, and delete anomalies.
    - Requires more complex application logic to maintain data consistency.
    - Consumes more storage space.
  - **Conclusion:** De-normalization is a trade-off. It should only be applied after thorough analysis and when performance benefits outweigh the risks to data integrity.

Session 4: SQL Data Definition Language (DDL), Data Manipulation Language (DML) & Data Control Language (DCL)

This session introduces the three main categories of SQL commands used to interact with a database: defining its structure (DDL), manipulating data within it (DML), and managing permissions (DCL). It also covers Transaction Control Language (TCL).

- **DDL Commands (Data Definition Language)**

  - **Purpose:** Used to define, modify, or delete database objects like tables, views, indexes, etc. DDL commands are auto-committed, meaning changes are permanent and cannot be rolled back.
  - **`CREATE`:** Used to create database objects.
    - `CREATE DATABASE database_name;`
    - `CREATE TABLE table_name ( column1 datatype PRIMARY KEY, column2 datatype UNIQUE, column3 datatype DEFAULT 'value', column4 datatype NOT NULL, column5 datatype, FOREIGN KEY (column5) REFERENCES other_table(other_column) );`
    - `CREATE INDEX index_name ON table_name (column1, column2);`
    - `CREATE VIEW view_name AS SELECT column1, column2 FROM table_name WHERE condition;`
  - **`ALTER`:** Used to modify the structure of existing database objects.
    - `ALTER TABLE table_name ADD column_name datatype;`
    - `ALTER TABLE table_name DROP COLUMN column_name;`
    - `ALTER TABLE table_name MODIFY COLUMN column_name new_datatype;`
    - `ALTER TABLE table_name RENAME COLUMN old_name TO new_name;`
    - `ALTER TABLE table_name RENAME TO new_table_name;`
  - **`DROP`:** Used to delete existing database objects.
    - `DROP DATABASE database_name;`
    - `DROP TABLE table_name;`

        - `DROP INDEX index_name ON table_name;`
        - `DROP VIEW view_name;`

- **DML Commands (Data Manipulation Language)**

    - **Purpose:** Used to manage data within the database objects. These commands are not auto-committed and can be rolled back.
    - **`SELECT`:** Used to retrieve data from one or more tables.
        - `SELECT column1, column2 FROM table_name WHERE condition ORDER BY column1 DESC;`
        - `SELECT * FROM table_name;`
    - **`INSERT`:** Used to add new rows of data into a table.
        - `INSERT INTO table_name (column1, column2) VALUES (value1, value2);`
        - `INSERT INTO table_name VALUES (value1, value2, value3);` (All columns)
    - **`UPDATE`:** Used to modify existing data in a table.
        - `UPDATE table_name SET column1 = new_value WHERE condition;`
    - **`DELETE`:** Used to remove one or more rows from a table.
        - `DELETE FROM table_name WHERE condition;`
        - `DELETE FROM table_name;` (Deletes all rows, but can be rolled back)
    - **`TRUNCATE`:** Used to remove all rows from a table. It's a DDL command in MySQL (implicitly commits), faster than `DELETE FROM table_name` for large tables because it logs less. It also resets auto-increment counters.
        - `TRUNCATE TABLE table_name;`

- **DCL Commands (Data Control Language)**

    - **Purpose:** Used to manage user permissions and access control to the database.
    - **`GRANT`:** Used to give specific privileges to users or roles.
        - `GRANT SELECT, INSERT ON database_name.table_name TO 'username'@'localhost';`
        - `GRANT ALL PRIVILEGES ON *.* TO 'adminuser'@'%' WITH GRANT OPTION;`
    - **`REVOKE`:** Used to remove granted privileges from users or roles.
        - `REVOKE INSERT ON database_name.table_name FROM 'username'@'localhost';`

- **TCL Commands (Transaction Control Language)**

    - **Purpose:** Used to manage transactions within the database. Transactions are a sequence of operations performed as a single logical unit of work. They ensure ACID properties (Atomicity, Consistency, Isolation, Durability).
    - **`COMMIT`:** Makes all changes performed in the current transaction permanent in the database.
        - `COMMIT;`
    - **`ROLLBACK`:** Undoes all changes performed in the current transaction, reverting the database to its state before the transaction began.
        - `ROLLBACK;`
    - **`SAVEPOINT`:** Creates a point within a transaction to which you can later roll back.
        - `SAVEPOINT savepoint_name;`
        - `ROLLBACK TO savepoint_name;`

Assignment – Lab (Session 4): SQL Practice Questions

Objective: Hands-on practice with DDL, DML, and TCL commands to create, manipulate, and control a simple database.

Commands to Practice:

- **DDL Commands:**
  - `CREATE DATABASE`
  - `CREATE TABLE` (with various data types, constraints like `PRIMARY KEY`, `NOT NULL`, `UNIQUE`, `DEFAULT`, `FOREIGN KEY`)
  - `ALTER TABLE` (add column, drop column, modify column datatype, rename column, rename table)
  - `DROP DATABASE`
  - `DROP TABLE`
  - `GRANT`
  - `REVOKE`
- **DML Commands:**
  - `SELECT` (simple queries, selecting all columns, specific columns)
  - `INSERT` (inserting full rows, partial rows)
  - `UPDATE` (updating single rows, multiple rows based on conditions)
  - `DELETE` (deleting single rows, multiple rows based on conditions)
  - `TRUNCATE TABLE`
- **TCL Commands:**
  - `ROLLBACK` (demonstrate with `INSERT`/`UPDATE`/`DELETE` operations)
  - `COMMIT` (save changes permanently)
  - *(Optional: `SAVEPOINT` for more advanced transaction control)*

Scenario Example:

1. **Create a Database:** `my_company_db`
2. **Create Tables:**
   - `Departments`: `dept_id` (PK), `dept_name` (UNIQUE), `location`
   - `Employees`: `emp_id` (PK), `first_name`, `last_name`, `email` (UNIQUE), `phone_number`, `hire_date`, `salary`, `dept_id` (FK referencing `Departments`), `manager_id` (FK referencing `Employees` itself for self-referencing).
3. **Insert Data:** Add at least 5 departments and 10 employees, ensuring some employees belong to the same department and some have managers.
4. **Practice DDL:**
   - Add a new column `status` (e.g., 'Active', 'Inactive') to `Employees` table.
   - Rename the `location` column in `Departments` to `dept_location`.
   - Change the `salary` column to support decimal values (e.g., `DECIMAL(10, 2)`).
   - Try to drop a table (then recreate it).
5. **Practice DML:**
   - Select all employees.

- Select employees with a salary greater than 50000.
- Update the salary of an employee.
- Update the department of several employees.
- Delete an employee.
- Demonstrate `TRUNCATE` (e.g., create a dummy table, insert data, truncate, then try to rollback).

6. **Practice TCL:**
   - Start a transaction. Insert a new employee. `ROLLBACK`. Verify the employee is not there.
   - Start a transaction. Update an employee's name. `COMMIT`. Verify the name change is permanent.
7. **Practice DCL (if permissions allow):**
   - Create a new user.
   - Grant `SELECT` privilege on the `Employees` table to the new user.
   - Test with the new user.
   - Revoke the `SELECT` privilege.

Session 5 & 6: Inbuilt Functions, Grouping, and Set Operators

These sessions cover essential SQL features for data manipulation and aggregation: using built-in functions, grouping data for summary, and combining query results.

- **Inbuilt Functions**
  - **Concept:** Predefined functions provided by the DBMS that perform specific operations on data values. They can be broadly categorized into single-row functions and aggregate (group) functions.
  - **Single-Row Functions (Scalar Functions):** Operate on individual rows and return a single result for each row.
    - **Number Functions:**
      - `ABS(number)`: Returns the absolute value.
        - `SELECT ABS(-10);` (Result: 10)
      - `CEIL(number)`: Returns the smallest integer greater than or equal to the number (round up).
        - `SELECT CEIL(5.2);` (Result: 6)
      - `FLOOR(number)`: Returns the largest integer less than or equal to the number (round down).
        - `SELECT FLOOR(5.9);` (Result: 5)
      - `ROUND(number, [decimals])`: Rounds a number to a specified number of decimal places.
        - `SELECT ROUND(5.678, 2);` (Result: 5.68)
      - `MOD(N, M)`: Returns the remainder of N divided by M.
        - `SELECT MOD(10, 3);` (Result: 1)
      - `POWER(base, exponent)` / `POW(base, exponent)`: Returns `base` raised to the power of `exponent`.
        - `SELECT POWER(2, 3);` (Result: 8)
    - **String Functions:**
      - `LENGTH(string)`: Returns the length of the string.

                - `UPPER(string)`: Converts string to uppercase.
                - `LOWER(string)`: Converts string to lowercase.
                - `SUBSTRING(string, start, length)` / `SUBSTR(string, start, length)`: Extracts a substring.
                - `CONCAT(string1, string2, ...)`: Concatenates two or more strings.
                - `TRIM([BOTH | LEADING | TRAILING] [char FROM] string)`: Removes leading/trailing characters.
            - **Date/Time Functions:**
                - `CURDATE()`: Returns the current date.
                - `NOW()`: Returns the current date and time.
                - `DATE_FORMAT(date, format)`: Formats a date/time value.
                - `DATEDIFF(date1, date2)`: Calculates the difference in days between two dates.
                - `YEAR(date)`, `MONTH(date)`, `DAY(date)`: Extract date parts.
            - **Control Flow Functions:**
                - `CASE WHEN condition THEN result [WHEN condition THEN result ...] ELSE result END`: Conditional logic.
                - `IF(condition, value_if_true, value_if_false)`: Simple conditional.
                - `COALESCE(value1, value2, ...)`: Returns the first non-NULL expression in the list.
                - `NULLIF(expr1, expr2)`: Returns NULL if `expr1 = expr2`, otherwise returns `expr1`.

    - **Group Value Functions (Aggregate Functions):** Operate on a set of rows (a group) and return a single summary value for that group.

        - `COUNT([DISTINCT] column | *)`: Counts the number of rows or non-NULL values.
            - `SELECT COUNT(*) FROM Employees;`
            - `SELECT COUNT(DISTINCT dept_id) FROM Employees;`
        - `SUM(column)`: Calculates the sum of values in a column.
            - `SELECT SUM(salary) FROM Employees;`
        - `AVG(column)`: Calculates the average of values in a column.
            - `SELECT AVG(salary) FROM Employees;`
        - `MAX(column)`: Returns the maximum value in a column.
            - `SELECT MAX(salary) FROM Employees;`
        - `MIN(column)`: Returns the minimum value in a column.
            - `SELECT MIN(salary) FROM Employees;`

- **Grouping Things Together (`GROUP BY`, `HAVING` Clause)**

    - `GROUP BY` **Clause:**
        - **Purpose:** Used with aggregate functions to group rows that have the same values in specified columns into a set of summary rows. The aggregate functions then operate on these groups.
        - **Placement:** Comes after the `FROM` and `WHERE` clauses, before `ORDER BY`.
        - **Syntax:** `SELECT column1, aggregate_function(column2) FROM table_name GROUP BY column1;`
        - **Example:** `SELECT dept_id, AVG(salary) FROM Employees GROUP BY dept_id;`
            - *(This query calculates the average salary for each department.)*
    - `HAVING` **Clause:**

    - **Purpose:** Used to filter the results of `GROUP BY` based on conditions applied to aggregate functions. It's similar to `WHERE`, but `WHERE` filters individual rows *before* grouping, while `HAVING` filters groups *after* aggregation.
    - **Placement:** Comes after the `GROUP BY` clause.
    - **Syntax:** `SELECT column1, aggregate_function(column2) FROM table_name GROUP BY column1 HAVING aggregate_condition;`
    - **Example:** `SELECT dept_id, AVG(salary) FROM Employees GROUP BY dept_id HAVING AVG(salary) > 60000;`
      - *(This query calculates the average salary for each department, but only shows departments where the average salary is greater than 60000.)*

- **Set Operators (`UNION`, `UNION ALL`)**
  - **Concept:** Used to combine the result sets of two or more `SELECT` statements into a single result set. For set operators to work, the `SELECT` statements must have the same number of columns, and corresponding columns must have compatible data types.
  - `UNION`:
    - **Purpose:** Combines the result sets and returns only distinct rows. It automatically removes duplicate rows.
    - **Example:**

```sql
SELECT first_name, last_name FROM Employees
UNION
SELECT contact_name, '' FROM Customers; -- Assuming Customers table exists
```

  - `UNION ALL`:
    - **Purpose:** Combines the result sets and returns all rows, including duplicates.
    - **Example:**

```sql
SELECT emp_id, first_name FROM Employees WHERE salary > 70000
UNION ALL
SELECT emp_id, first_name FROM Employees WHERE dept_id = 10;
```

    (If an employee earns >70k and is in dept 10, they'll appear twice.)
  - **Other Set Operators (Conceptual, MySQL does not have direct support for these as operators but they can be simulated):**
    - `INTERSECT`: Returns only the rows that are common to both result sets (distinct rows).
      - *Simulation in MySQL:* Using `INNER JOIN` or `IN` with a subquery.
    - `EXCEPT` **(or** `MINUS`**):** Returns the rows that are in the first result set but not in the second (distinct rows).
      - *Simulation in MySQL:* Using `LEFT JOIN` with `WHERE column IS NULL` or `NOT IN` with a subquery.

Assignment – Lab (Session 5 & 6): SQL Functions and Set Operators

Objective: Apply various built-in SQL functions and set operators to extract, transform, and combine data effectively.

Commands to Practice:

- **Number Functions:**
  - `ABS`, `CEIL`, `FLOOR`, `ROUND`, `MOD`, `POWER` (or `POW`).
  - *Example:* Calculate the absolute difference between salaries, round average salaries.
- **Single Value Functions (String, Date/Time, Control Flow):**
  - `LENGTH`, `UPPER`, `LOWER`, `SUBSTRING`, `CONCAT`, `TRIM`.
  - `CURDATE()`, `NOW()`, `DATE_FORMAT`, `DATEDIFF`, `YEAR`, `MONTH`, `DAY`.
  - `CASE`, `IF`, `COALESCE`, `NULLIF`.
  - *Example:* Display employee names in uppercase, get employee age, create a formatted report date, handle NULL manager IDs.
- **Group Value Functions (Aggregate Functions):**
  - `AVG`, `COUNT`, `MAX`, `MIN`, `SUM`.
  - *Example:* Find total salary, average salary per department, number of employees in each department, highest/lowest salary.
- **Queries to cover `GROUP BY` and `HAVING`:**
  - Find the number of employees in each department.
  - Find departments with an average salary greater than 60000.
  - Find the maximum salary in each department that has more than 5 employees.
- **Queries to cover Set Operators:**
  - `UNION`: Combine lists of names from `Employees` and another table (e.g., `Customers`) ensuring distinct names.
  - `UNION ALL`: Combine lists of active and inactive employees, showing all occurrences.
  - *(Optional: Attempt to simulate `INTERSECT` and `EXCEPT` using `JOIN` or subqueries if comfortable.)*

Scenario Example (using `my_company_db` from previous session):

1. **Functions:**
   - Display `first_name` and `last_name` of all employees as a single `Full_Name` column in uppercase.
   - Show the length of each employee's `email`.
   - Calculate the number of years each employee has been working (`DATEDIFF` between `NOW()` and `hire_date`).
   - If an employee's `phone_number` is `NULL`, display 'N/A'. (Use `COALESCE`).
   - Create a report showing 'High Salary' if `salary > 70000`, 'Medium Salary' if `salary` between 40000 and 70000, else 'Low Salary'. (Use `CASE`).
2. **Grouping:**
   - Get the count of employees in each `dept_id`.
   - Find the maximum and minimum salaries for each `dept_id`.
   - Calculate the total salary budget for each `dept_id`.
   - List departments where the average salary is below 50000.
   - Find departments that have more than 3 employees, and also show their average salary.
3. **Set Operators:**

  - Combine the `first_name` of employees and `dept_name` of departments into one list using `UNION`. (Make sure they have compatible columns, e.g., add `NULL` for `last_name` in one, or select only first name).
  - Combine `emp_id` and `first_name` for employees hired before 2020 and employees in `dept_id` 10, using `UNION ALL`.

Session 7: Filtering, Sorting, and Relational Algebra

This session focuses on advanced filtering techniques, sorting query results, and introduces the theoretical foundations of relational databases through Relational Algebra.

- **`LIKE` Operator**

  - **Purpose:** Used in the `WHERE` clause to search for a specified pattern in a column.
  - **Wildcard Characters:**
    - `%`: Represents zero, one, or multiple characters.
    - `_`: Represents a single character.
  - **Syntax:** `SELECT columns FROM table WHERE column LIKE 'pattern';`
  - **Examples:**
    - `SELECT * FROM Employees WHERE first_name LIKE 'A%';` (Starts with 'A')
    - `SELECT * FROM Employees WHERE last_name LIKE '%son';` (Ends with 'son')
    - `SELECT * FROM Employees WHERE email LIKE '%@example.com%';` (Contains '@example.com')
    - `SELECT * FROM Employees WHERE phone_number LIKE '___-___-____';` (Matches specific format)
    - `SELECT * FROM Employees WHERE first_name NOT LIKE 'A%';` (Does not start with 'A')

- **`DISTINCT`**

  - **Purpose:** Used in the `SELECT` statement to eliminate duplicate rows from the result set. It applies to all selected columns.
  - **Syntax:** `SELECT DISTINCT column1, column2 FROM table_name;`
  - **Example:** `SELECT DISTINCT dept_id FROM Employees;` (Lists unique department IDs)
  - `SELECT DISTINCT city, state FROM Customers;` (Lists unique combinations of city and state)

- **Sorting (`ORDER BY` clause)**

  - **Purpose:** Used to sort the result set of a query in ascending or descending order based on one or more columns.
  - **Syntax:** `SELECT columns FROM table ORDER BY column1 [ASC|DESC], column2 [ASC|DESC];`
  - **`ASC` (Ascending):** Default order, from lowest to highest.
  - **`DESC` (Descending):** From highest to lowest.
  - **Examples:**
    - `SELECT first_name, last_name, salary FROM Employees ORDER BY salary DESC;`

    - `SELECT first_name, last_name FROM Employees ORDER BY last_name ASC, first_name ASC;` (Sorts by last name, then by first name for ties)

- **`BETWEEN`, `AND`, `OR` Operators**
  - **`BETWEEN`:**
    - **Purpose:** Filters results within a specified range (inclusive). Can be used for numbers, dates, and strings.
    - **Syntax:** `SELECT columns FROM table WHERE column BETWEEN value1 AND value2;`
    - **Example:** `SELECT * FROM Employees WHERE salary BETWEEN 50000 AND 70000;`
    - `SELECT * FROM Orders WHERE order_date BETWEEN '2023-01-01' AND '2023-03-31';`
  - **`AND`:**
    - **Purpose:** A logical operator that combines multiple conditions in the `WHERE` clause. Both conditions must be true for the row to be selected.
    - **Syntax:** `SELECT columns FROM table WHERE condition1 AND condition2;`
    - **Example:** `SELECT * FROM Employees WHERE dept_id = 10 AND salary > 60000;`
  - **`OR`:**
    - **Purpose:** A logical operator that combines multiple conditions in the `WHERE` clause. If at least one of the conditions is true, the row is selected.
    - **Syntax:** `SELECT columns FROM table WHERE condition1 OR condition2;`
    - **Example:** `SELECT * FROM Employees WHERE dept_id = 10 OR dept_id = 20;`

- **Comparing Nulls (`IS NULL`/`IS NOT NULL`)**
  - **Concept:** `NULL` represents the absence of a value. It's not equivalent to zero or an empty string. Standard comparison operators (`=`, `!=`, `<`, `>`) do not work reliably with `NULL`.
  - **`IS NULL`:**
    - **Purpose:** Checks if a column's value is `NULL`.
    - **Syntax:** `SELECT columns FROM table WHERE column IS NULL;`
    - **Example:** `SELECT * FROM Employees WHERE manager_id IS NULL;` (Finds employees without a manager)
  - **`IS NOT NULL`:**
    - **Purpose:** Checks if a column's value is *not* `NULL`.
    - **Syntax:** `SELECT columns FROM table WHERE column IS NOT NULL;`
    - **Example:** `SELECT * FROM Employees WHERE phone_number IS NOT NULL;`

- **`IN`/`NOT IN`**
  - **`IN`:**
    - **Purpose:** Checks if a column's value matches any value in a list of specified values or in the result set of a subquery. It's a shorthand for multiple `OR` conditions.
    - **Syntax:** `SELECT columns FROM table WHERE column IN (value1, value2, ...);`
    - **Example:** `SELECT * FROM Employees WHERE dept_id IN (10, 20, 30);`
    - `SELECT * FROM Employees WHERE dept_id IN (SELECT dept_id FROM Departments WHERE location = 'New York');`
  - **`NOT IN`:**

        - **Purpose:** Checks if a column's value does *not* match any value in a list or subquery result. Shorthand for multiple `AND NOT` conditions.
        - **Syntax:** `SELECT columns FROM table WHERE column NOT IN (value1, value2, ...);`
        - **Example:** `SELECT * FROM Employees WHERE dept_id NOT IN (10, 20);`

- **Relational Algebra Operations (Selection, Projection, Union, Intersect, Minus)**
    - **Concept:** A procedural query language that operates on relations (tables) and produces relations as output. It forms the theoretical basis for SQL and helps understand how queries are processed.
    - **Selection (σ - sigma):**
        - **Purpose:** Filters rows (tuples) from a relation based on a specified condition. Analogous to the `WHERE` clause in SQL.
        - **Notation:** `σ condition (Relation)`
        - **Example:** `σ salary > 50000 (Employees)`
    - **Projection (π - pi):**
        - **Purpose:** Filters columns (attributes) from a relation, returning a new relation with a subset of the original columns. Analogous to selecting specific columns in the `SELECT` clause in SQL. Duplicates are typically eliminated in true relational algebra.
        - **Notation:** `π column1, column2 (Relation)`
        - **Example:** `π first_name, last_name (Employees)`
    - **Union (∪):**
        - **Purpose:** Combines all tuples from two relations, removing duplicates. Both relations must be "union-compatible" (same number of columns and compatible data types). Analogous to `UNION` in SQL.
        - **Notation:** `Relation1 ∪ Relation2`
        - **Example:** `π name (Students) ∪ π name (Faculty)`
    - **Intersection (∩):**
        - **Purpose:** Returns tuples that are common to both relations. Both relations must be union-compatible. Analogous to `INTERSECT` (not directly in MySQL).
        - **Notation:** `Relation1 ∩ Relation2`
        - **Example:** `π email (Employees) ∩ π email (Customers)`
    - **Set Difference (Minus, -):**
        - **Purpose:** Returns tuples that are in the first relation but not in the second. Both relations must be union-compatible. Analogous to `EXCEPT` or `MINUS` (not directly in MySQL).
        - **Notation:** `Relation1 - Relation2`
        - **Example:** `π product_id (AllProducts) - π product_id (SoldProducts)`
    - **Cartesian Product (× - cross product):**
        - **Purpose:** Combines every tuple from the first relation with every tuple from the second relation. If R1 has `n` rows and R2 has `m` rows, the result has `n * m` rows. Often the first step before a join. Analogous to `CROSS JOIN` in SQL.
        - **Notation:** `Relation1 × Relation2`
    - **Join (⋈ - natural join):**
        - **Purpose:** Combines tuples from two relations based on a common attribute or a join condition. It's a combination of a Cartesian product, selection, and projection. (More details in later session).
        - **Notation:** `Relation1 ⋈ Relation2`

Assignment – Lab (Session 7): Filtering and Relational Operators

Objective: Enhance query writing skills using advanced filtering, sorting, and comparison operators.

Commands to Practice:

- **Using `LIKE`:**
  - Find all employees whose `first_name` starts with 'J'.
  - Find all employees whose `email` contains 'gmail'.
  - Find all departments whose `dept_name` ends with 'ing'.
- **Using `DISTINCT`:**
  - List all unique `location` values from the `Departments` table.
  - List all unique combinations of `dept_id` and `salary` from the `Employees` table.
- **Using `ORDER BY`:**
  - List all employees, sorted by `salary` in descending order.
  - List all employees, sorted by `dept_id` ascending, then by `last_name` ascending.
- **Using `BETWEEN...AND`:**
  - Find employees with `salary` between 55000 and 75000 (inclusive).
  - Find employees hired between '2020-01-01' and '2022-12-31'.
- **Using `AND`, `OR`:**
  - Find employees in `dept_id` 20 AND `salary` > 60000.
  - Find employees in `dept_id` 10 OR `dept_id` 30.
  - Combine conditions: Employees in `dept_id` 10 with `salary` > 50000, OR employees in `dept_id` 20.
- **Comparing Nulls (`IS NULL`/`IS NOT NULL`):**
  - List all employees who do NOT have a `phone_number`.
  - List all employees who have a `manager_id`.
- **Using `IN`/`NOT IN`:**
  - Find employees whose `dept_id` is 10, 20, or 40.
  - Find employees whose `dept_id` is NOT 10 or 20.
  - Find employees whose `dept_id` is one of the departments located in 'Chicago'. (Requires a subquery).

Session 8 & 9: SQL Functions (Revisited) and Advanced Grouping

These sessions reinforce the use of SQL functions and `GROUP BY`/`HAVING` with more complex scenarios and combinations, providing deeper mastery.

- **SQL Functions (Reinforcement & Advanced Usage)**

  - **Review of Single-Row Functions:** Revisit `ABS`, `CEIL`, `FLOOR`, `ROUND`, `MOD`, `POWER`, `LENGTH`, `UPPER`, `LOWER`, `SUBSTRING`, `CONCAT`, `TRIM`, `CURDATE()`, `NOW()`, `DATE_FORMAT`, `DATEDIFF`, `YEAR`, `MONTH`, `DAY`.
  - **Common Advanced Use Cases:**
    - Combining functions: `SELECT UPPER(CONCAT(first_name, ' ', last_name)) AS FullName FROM Employees;`
    - Conditional formatting using `CASE`:

```sql
SELECT first_name, salary,
       CASE
           WHEN salary > 70000 THEN 'High Earner'
           WHEN salary >= 50000 THEN 'Mid Earner'
           ELSE 'Low Earner'
       END AS SalaryCategory
FROM Employees;
```

- Handling potential `NULL`s in calculations:

```sql
-- Calculate bonus as 10% of salary, if salary is NULL, bonus is 0
SELECT first_name, salary, COALESCE(salary * 0.10, 0) AS Bonus
FROM Employees;
```

- Converting data types for specific operations (e.g., `CAST` or `CONVERT` in MySQL).

- **Aggregate Functions (Reinforcement & Advanced Usage)**
  - **Review of `AVG`, `COUNT`, `MAX`, `MIN`, `SUM`.**
  - **`COUNT(DISTINCT column)`:** Counting unique values within a group.
    - `SELECT dept_id, COUNT(DISTINCT location) FROM Departments GROUP BY dept_id;` (Less meaningful here, but illustrates concept.)
  - **Handling NULLs in Aggregates:** Most aggregate functions (except `COUNT(*)`) ignore `NULL` values.
    - `SELECT AVG(salary) FROM Employees;` (Only averages non-NULL salaries).
  - **Using `WITH ROLLUP` (MySQL Specific):**
    - **Purpose:** Generates super-aggregate rows along with the regular grouped rows. Adds subtotals and a grand total.
    - **Syntax:** `SELECT column1, column2, aggregate_function(column3) FROM table_name GROUP BY column1, column2 WITH ROLLUP;`
    - **Example:**

```sql
SELECT dept_id, manager_id, COUNT(*) AS EmployeeCount
FROM Employees
GROUP BY dept_id, manager_id WITH ROLLUP;
```

(This will show counts for each dept_id/manager_id pair, then subtotals for each dept_id, and a grand total.)

- **Grouping Things Together (`GROUP BY`, `HAVING` - Advanced)**
  - **Multiple Columns in `GROUP BY`:**
    - Group by more than one column to create finer-grained groups.
    - `SELECT dept_id, manager_id, COUNT(*) FROM Employees GROUP BY dept_id, manager_id;` *(This counts employees for each unique combination of department and*

        manager.)
    - **Combining `WHERE` and `HAVING`:**
        - Remember the order: `FROM` -> `WHERE` -> `GROUP BY` -> `HAVING` -> `SELECT` -> `ORDER BY`.
        - `WHERE` filters individual rows *before* grouping and aggregation.
        - `HAVING` filters groups *after* grouping and aggregation.
        - **Example:**

```
SELECT dept_id, AVG(salary) AS AvgSalary
FROM Employees
WHERE hire_date > '2020-01-01' -- Filter individual employees
hired after 2020
GROUP BY dept_id
HAVING COUNT(*) > 2; -- Filter groups that have more than 2
employees
```

    - **Common Mistakes with `GROUP BY`:**
        - Not including non-aggregated columns in the `GROUP BY` clause (except in MySQL where this is allowed as an extension, but leads to ambiguous results and should be avoided in strict SQL).
        - Using aggregate functions in `WHERE` clause (use `HAVING` instead).

Session 10 & 11: Subqueries & Joins

These sessions are critical for combining data from multiple tables and performing complex data retrieval using both subqueries and various types of joins.

- **Subqueries (Nested Queries)**

    - **Concept:** A query (inner query or inner select) embedded inside another SQL query (outer query). The inner query executes first, and its result is used by the outer query.
    - **Types of Subqueries based on Result:**
        - **Scalar Subquery:** Returns a single value (one row, one column). Can be used anywhere a single value is expected (e.g., `SELECT` list, `WHERE` clause, `SET` clause in `UPDATE`).
            - `SELECT first_name, (SELECT AVG(salary) FROM Employees) AS OverallAvgSalary FROM Employees;`
        - **Row Subquery:** Returns a single row (multiple columns). Can be used in `WHERE` or `HAVING` clauses, often with operators like `=`, `!=`, `>`, `<`, etc.
            - `SELECT * FROM Employees WHERE (dept_id, salary) = (SELECT dept_id, salary FROM Employees WHERE emp_id = 101);`
        - **Table Subquery:** Returns a table (multiple rows, multiple columns). Can be used in the `FROM` clause (derived table), `IN`/`EXISTS` operators.
            - `SELECT dept_name FROM Departments WHERE dept_id IN (SELECT dept_id FROM Employees WHERE salary > 80000);`
    - **Usage in different clauses:**
        - **`WHERE` clause:** Most common. Filters rows based on a condition involving the subquery result.

            - `SELECT * FROM Employees WHERE salary > (SELECT AVG(salary) FROM Employees);`
        - `FROM` **clause (Derived Tables/Inline Views):** The subquery acts as a temporary, named table that can be joined with other tables.
            - `SELECT d.dept_name, e.AvgDeptSalary FROM Departments d JOIN (SELECT dept_id, AVG(salary) AS AvgDeptSalary FROM Employees GROUP BY dept_id) e ON d.dept_id = e.dept_id;`
        - `SELECT` **clause:** As a scalar subquery to fetch a single value for each row.
            - `SELECT emp_id, first_name, (SELECT dept_name FROM Departments d WHERE d.dept_id = e.dept_id) AS DepartmentName FROM Employees e;`
        - `HAVING` **clause:** Used to filter groups based on a subquery's result.
            - `SELECT dept_id, COUNT(*) FROM Employees GROUP BY dept_id HAVING AVG(salary) > (SELECT AVG(salary) FROM Employees);`
    - **Correlated vs. Non-Correlated Subqueries:**
        - **Non-Correlated (Independent):** The inner query executes completely independently of the outer query. It runs once and passes its result to the outer query. (All examples above are non-correlated).
        - **Correlated:** The inner query depends on the outer query for its values and executes once for *each row* processed by the outer query. Often used with `EXISTS`, `NOT EXISTS`, `IN`, `NOT IN`.
            - `SELECT e.first_name, e.salary FROM Employees e WHERE e.salary = (SELECT MAX(e2.salary) FROM Employees e2 WHERE e2.dept_id = e.dept_id);`
            - *(This finds employees who earn the highest salary within their own department.)*

- **Joins**

    - **Concept:** Used to combine rows from two or more tables based on a related column between them. Joins are fundamental for retrieving data stored in a normalized relational database.
    - **Syntax (Standard `JOIN`):** `SELECT columns FROM table1 JOIN_TYPE table2 ON join_condition;`
    - **Types of Joins:**
        1. **Inner Join (`INNER JOIN` or simply `JOIN`):**
            - **Purpose:** Returns only the rows that have matching values in *both* tables based on the join condition. It's the most common type of join.
            - **Example:**

              ```
              SELECT e.first_name, d.dept_name
              FROM Employees e
              INNER JOIN Departments d ON e.dept_id = d.dept_id;
              ```

              (Only employees who have a matching department ID in the Departments table will be returned.)
        2. **Outer Joins:** Return matching rows *plus* rows from one table that don't have matches in the other.
            - `LEFT JOIN` **(or** `LEFT OUTER JOIN`**):**

        - **Purpose:** Returns all rows from the *left* table and the matching rows from the *right* table. If there's no match in the right table, NULLs are returned for the right table's columns.
        - **Example:**

          ```sql
          SELECT e.first_name, d.dept_name
          FROM Employees e
          LEFT JOIN Departments d ON e.dept_id = d.dept_id;
          ```

          (Returns all employees, including those without a department, showing NULL for dept_name.)
    - `RIGHT JOIN` **(or** `RIGHT OUTER JOIN`**):**
        - **Purpose:** Returns all rows from the *right* table and the matching rows from the *left* table. If there's no match in the left table, NULLs are returned for the left table's columns.
        - **Example:**

          ```sql
          SELECT e.first_name, d.dept_name
          FROM Employees e
          RIGHT JOIN Departments d ON e.dept_id = d.dept_id;
          ```

          (Returns all departments, including those with no employees, showing NULL for employee details.)
    - `FULL JOIN` **(or** `FULL OUTER JOIN`**):**
        - **Purpose:** Returns all rows when there is a match in one of the tables. It effectively combines the results of `LEFT JOIN` and `RIGHT JOIN`, including non-matching rows from both sides.
        - **Note:** MySQL does not directly support `FULL OUTER JOIN`. It can be simulated using `LEFT JOIN UNION ALL RIGHT JOIN` with conditions to exclude duplicates.
3. `NATURAL JOIN`**:**
    - **Purpose:** Joins two tables based on all columns that have the same name and data type in both tables. It automatically finds the common columns.
    - **Syntax:** `SELECT columns FROM table1 NATURAL JOIN table2;`
    - **Caution:** Can be unpredictable if columns with the same name exist but are not intended for joining.
    - **Example:** `SELECT * FROM Employees NATURAL JOIN Departments;` (Assumes `dept_id` is common)
4. `CROSS JOIN`**:**
    - **Purpose:** Produces a Cartesian product of the two tables. Every row from the first table is combined with every row from the second table. No join condition is specified.
    - **Syntax:** `SELECT columns FROM table1 CROSS JOIN table2;` (Equivalent to `SELECT ... FROM table1, table2;`)

        - **Example:** `SELECT e.first_name, d.dept_name FROM Employees e CROSS JOIN Departments d;` *(If Employees has 100 rows and Departments has 10, this will return 1000 rows.)*

- **Self Join:**
    - **Concept:** A table is joined with itself. Used when you need to compare rows within the same table, often when there's a hierarchical relationship (e.g., manager-employee).
    - **Example:**

    ```sql
    SELECT e.first_name AS Employee, m.first_name AS Manager
    FROM Employees e
    JOIN Employees m ON e.manager_id = m.emp_id;
    ```

- **Sub Query Joins (Derived Table Joins)**
    - **Concept:** Using the result of a subquery as a table in a join operation. The subquery is often placed in the `FROM` clause and given an alias.
    - **Purpose:** Allows you to perform complex aggregations or filters *before* joining, or to simplify complex multi-table queries.
    - **Example:**

    ```sql
    SELECT d.dept_name, a.AvgSalary
    FROM Departments d
    JOIN (SELECT dept_id, AVG(salary) AS AvgSalary FROM Employees GROUP BY
    dept_id) AS a
    ON d.dept_id = a.dept_id
    WHERE a.AvgSalary > 60000;
    ```

    (This query first calculates average salaries per department in a subquery, then joins that result with the Departments table.)

Assignment – Lab (Session 10 & 11): Joins & Subqueries

Objective: Master the use of various join types and subqueries to retrieve complex datasets from a relational database.

Commands to Practice:

- **SQL queries on Joins covering all types of Joins:**
    - **Inner Join:** List employees and their department names.
    - **Left Join:** List all departments, and any employees in them. Include departments with no employees.
    - **Right Join:** List all employees, and their department names. Include employees with no department. (Note: In MySQL, `RIGHT JOIN` can be re-written as `LEFT JOIN` by swapping table order).

  - **Self Join:** List all employees along with their manager's name.
  - **Cross Join:** Perform a cross join between `Employees` and `Departments` (understand its purpose/result).
  - **Using `WHERE` clause for joining (implicit join):** Re-write an `INNER JOIN` using the `FROM table1, table2 WHERE condition` syntax.
- **SQL Practice Questions covering Correlated Queries, Sub Queries:**
  - **Scalar Subquery:**
    - Find employees whose salary is greater than the overall average salary of all employees.
    - Select each employee's name and the number of employees in their department.
  - **Table Subquery (in `WHERE` or `IN`):**
    - Find employees who work in departments located in 'New York'.
    - Find departments that have no employees. (Using `NOT IN` or `NOT EXISTS`).
  - **Derived Tables (Subquery in `FROM`):**
    - Find the department with the highest average salary.
    - List the top 3 highest-earning employees in each department.
  - **Correlated Queries:**
    - Find the names of employees who earn more than the average salary in their own department.
    - Find departments that have at least one employee earning more than 80000. (Using `EXISTS`).

Scenario Example (using `my_company_db`):

1. **Joins:**
   - Retrieve `employee_id`, `first_name`, `last_name`, `salary`, and `department_name` for all employees.
   - Retrieve `department_name` and the `first_name` of employees. Include departments that currently have no employees.
   - List all employees and their manager's `first_name` (if they have one).
   - List all department names and the count of employees in each department.
2. **Subqueries:**
   - Find the `first_name` and `salary` of employees who earn more than the employee with `emp_id = 105`.
   - List the `dept_name` of departments that have at least one employee whose `salary` is `NULL`.
   - Find the `first_name` and `salary` of employees who have the minimum salary in their respective department.
   - List all departments that do not have any employees (using a subquery).

Session 12: Views, Indexes, Temporary Tables, and Database Locks

This session introduces advanced database objects and concepts crucial for database performance, security, and concurrency.

- **Views, Types of Views, Simple and Complex Views**

  - **Concept:** A view is a virtual table based on the result set of an SQL query. It does not store data itself but instead provides a logical representation of data stored in underlying base tables.

- **Benefits of Views:**
  - **Security:** Can restrict access to specific rows and columns without granting direct access to the base tables.
  - **Simplicity:** Hides complex queries (e.g., joins, aggregations) behind a simpler table interface.
  - **Logical Data Independence:** Provides a consistent interface even if the underlying base tables change (as long as the view definition can adapt).
  - **Reusability:** Common queries can be saved as views and reused.
- **Types of Views:**
  - **Simple View:**
    - Based on a single base table.
    - Does not contain aggregate functions, `GROUP BY`, `DISTINCT`, or expressions.
    - Typically **updatable** (inserts, updates, deletes on the view affect the underlying table).
    - **Example:** `CREATE VIEW Emp_Basic_Info AS SELECT emp_id, first_name, last_name FROM Employees WHERE salary > 50000;`
  - **Complex View:**
    - Based on multiple tables (joins), or contains aggregate functions, `GROUP BY`, `DISTINCT`, `UNION`, or expressions.
    - Generally **not updatable** (inserts, updates, deletes on the view are not supported or are highly restricted because the DBMS cannot unambiguously map changes back to the base tables).
    - **Example:** `CREATE VIEW Dept_Avg_Salary AS SELECT d.dept_name, AVG(e.salary) AS AvgSalary FROM Departments d JOIN Employees e ON d.dept_id = e.dept_id GROUP BY d.dept_name;`
- **`CREATE VIEW`, `ALTER VIEW`, `DROP VIEW` Commands:**
  - `CREATE VIEW view_name AS SELECT ...;`
  - `ALTER VIEW view_name AS SELECT ...;` (To modify an existing view)
  - `DROP VIEW view_name;`

- **Indexes, Benefit of Indexes, Type of Indexes**

  - **Concept:** A special lookup table that the database search engine can use to speed up data retrieval. Think of it like an index in a book. It contains a list of values from the indexed column(s) and pointers to the physical location of the corresponding rows.
  - **Benefit of Indexes:**
    - **Faster Data Retrieval:** Significantly reduces the time required to find specific rows, especially for `SELECT` queries with `WHERE` clauses, `JOIN` conditions, and `ORDER BY` clauses.
    - **Improved Query Performance:** Leads to faster response times for users and applications.
  - **Drawbacks of Indexes:**
    - **Slower Data Modification:** `INSERT`, `UPDATE`, and `DELETE` operations on indexed columns become slower because the index itself must also be updated.
    - **Disk Space Consumption:** Indexes require additional storage space.
    - **Overhead:** Requires management overhead.
  - **Type of Indexes (in MySQL):**
    - **Primary Key Index:** Automatically created when a `PRIMARY KEY` constraint is defined on a table. It's a unique index and often clustered (see below).

- **Unique Index:** Ensures that all values in the indexed column(s) are unique. Prevents duplicate entries.
  - `CREATE UNIQUE INDEX UIdx_Email ON Employees (email);`
- **Clustered Index:** (Specific to Storage Engine, e.g., InnoDB in MySQL uses primary key as clustered index).
  - **Concept:** Determines the physical order in which data rows are stored on disk. A table can have only one clustered index.
  - **Benefit:** Excellent for range queries and retrieving data in sorted order.
- **Non-Clustered Index (Secondary Index):**
  - **Concept:** A separate structure from the data rows. It stores the indexed columns' values and pointers to the actual data rows. A table can have multiple non-clustered indexes.
  - **Benefit:** Good for specific lookups.
  - `CREATE INDEX Idx_LastName ON Employees (last_name);`
- **Composite Index (Multi-column Index):** An index on two or more columns. Useful for queries that filter or sort on multiple columns.
  - `CREATE INDEX Idx_DeptSal ON Employees (dept_id, salary);`
- **Full-Text Index:** Specifically designed for efficient text searches on large blocks of text data (e.g., using `MATCH AGAINST`).

- **Temporary Tables**

  - **Concept:** Tables that exist only for the duration of a specific database session or transaction. They are automatically dropped when the session ends or the transaction is committed/rolled back (depending on `ON COMMIT` action).

  - **Benefits:**

    - **Break Down Complex Queries:** Can store intermediate results of complex queries, making them easier to manage and debug.
    - **Performance Optimization:** Sometimes, storing intermediate results in a temporary table and then querying it can be faster than a single, very complex query.
    - **Isolation:** Changes to temporary tables do not affect other sessions or the permanent database.

  - **Syntax:** `CREATE TEMPORARY TABLE temp_table_name (column definitions) SELECT ...;`

  - **Example:**

```sql
CREATE TEMPORARY TABLE HighEarners AS
SELECT emp_id, first_name, salary
FROM Employees
WHERE salary > 70000;

SELECT * FROM HighEarners;
```

- **Database Locks**

- **Concept:** Mechanisms used by a DBMS to control concurrent access to data, ensuring data consistency and integrity, especially in multi-user environments. When a transaction acquires a lock on a data item, other transactions are prevented from modifying (or sometimes reading) that item until the lock is released.
- **Purpose:** To prevent concurrency issues like lost updates, dirty reads, non-repeatable reads, and phantom reads.
- **Types of Locks:**
  - **Shared Lock (S-Lock / Read Lock):** Allows multiple transactions to read the same data item concurrently. No transaction can acquire an exclusive lock on the item while shared locks exist.
  - **Exclusive Lock (X-Lock / Write Lock):** Only one transaction can hold an exclusive lock on a data item at a time. No other transaction can acquire any type of lock (shared or exclusive) on that item while an exclusive lock exists.
- **Lock Granularity:** The size of the data item being locked.
  - **Row-level Locking:** Locks individual rows. Offers highest concurrency but incurs more overhead. (Common in InnoDB).
  - **Page-level Locking:** Locks a block of rows (a page). Balance between concurrency and overhead.
  - **Table-level Locking:** Locks the entire table. Simplest to implement, but lowest concurrency. (Common in MyISAM).
  - **Database-level Locking:** Locks the entire database. Very low concurrency.
- **Implicit vs. Explicit Locking:**
  - **Implicit:** The DBMS automatically acquires and releases locks during transactions (most common).
  - **Explicit:** Developers manually issue `LOCK TABLES` or `UNLOCK TABLES` (less common, can lead to deadlocks if not used carefully).
- **Deadlocks:** A situation where two or more transactions are permanently blocked because each is waiting for a resource that is held by another transaction in the same set. DBMS typically have deadlock detection and resolution mechanisms (e.g., roll back one of the transactions).

Assignment – Lab (Session 12): Views, Indexes, Temporary Tables

Objective: Gain practical experience with creating and managing views, indexes, and temporary tables, and understanding the concept of database locks.

Commands to Practice:

- **Creating Views:**
  - Create a **simple view** `Employee_View` that shows `emp_id`, `first_name`, `last_name`, and `email` for all employees.
  - Query `Employee_View`.
  - Try to `INSERT`, `UPDATE`, `DELETE` through `Employee_View`. (Observe success for simple views).
  - Create a **complex view** `Dept_Employee_Count_View` showing `dept_name` and the count of employees in each department.
  - Query `Dept_Employee_Count_View`.
  - Try to `INSERT`, `UPDATE`, `DELETE` through `Dept_Employee_Count_View`. (Observe failure or restrictions for complex views).

- `ALTER` an existing view (e.g., add `phone_number` to `Employee_View`).
- `DROP` a view.

- **Creating Indexes:**
    - Create a non-unique index on the `last_name` column of the `Employees` table.
    - Create a unique index on the `phone_number` column of the `Employees` table. (Try inserting a duplicate value and observe the error).
    - Create a composite index on `dept_id` and `salary` columns of the `Employees` table.
    - `SHOW INDEXES FROM Employees;` (To verify created indexes).
    - `DROP INDEX` (e.g., `DROP INDEX UIdx_PhoneNumber ON Employees;`).
- **Creating Temporary Tables:**
    - Create a temporary table `Temp_HighSalariedEmployees` by selecting `emp_id`, `first_name`, `salary` from `Employees` where `salary > 70000`.
    - Query `Temp_HighSalariedEmployees`.
    - Try to query it from another session. (It should not be visible).
    - Observe that it disappears when your session ends.
- **Database Locks (Conceptual Understanding & Basic Demonstration):**
    - Discuss the concept of row-level vs. table-level locks.
    - **Demonstrate Implicit Locking (using two separate MySQL client sessions):**
        - **Session 1:** `START TRANSACTION; UPDATE Employees SET salary = 90000 WHERE emp_id = 101;` (Do NOT `COMMIT` or `ROLLBACK`).
        - **Session 2:** `SELECT salary FROM Employees WHERE emp_id = 101;` (Should see old value). Then `UPDATE Employees SET salary = 95000 WHERE emp_id = 101;` (Session 2 should block, waiting for Session 1's lock).
        - **Session 1:** `COMMIT;` (Session 2 should then unblock and complete).
        - Discuss how deadlocks can occur and how the DBMS usually handles them.

Session 13: Stored Procedures

This session introduces stored procedures, which are pre-compiled SQL code blocks that can enhance database functionality, performance, and security.

- **Introduction to Stored Procedures**

    - **Concept:** A stored procedure is a prepared SQL code that you can save and reuse. It's a set of SQL statements that perform a specific task, often taking input parameters and returning output parameters.
    - **Benefits of Stored Procedures:**
        - **Performance:**
            - **Reduced Network Traffic:** Instead of sending multiple SQL statements, only the procedure call is sent.
            - **Faster Execution:** Procedures are pre-compiled and stored on the database server, leading to faster execution times.
            - **Caching:** The database often caches the execution plan for procedures.
        - **Security:** Users can be granted permission to execute procedures without having direct access to the underlying tables, enhancing data security.
        - **Reusability:** Procedures can be called multiple times by various applications, promoting code reuse and reducing development effort.

        - **Modularity:** Complex tasks can be broken down into smaller, manageable procedures.
        - **Data Consistency & Integrity:** Centralizing business logic in procedures ensures consistent data manipulation across all applications.
        - **Maintainability:** Easier to maintain and modify business logic in one place (the procedure) rather than scattered across application code.

- **Procedure Parameters (`IN`, `OUT` and `INOUT`)**

    - Stored procedures can accept parameters, allowing them to be flexible and reusable for different inputs.
    - `IN` **Parameter (Default):**
        - **Purpose:** An input parameter. The calling program passes a value to the procedure. The procedure can use this value but cannot modify it or return a changed value to the caller.
        - **Example:** `IN emp_id_param INT`
    - `OUT` **Parameter:**
        - **Purpose:** An output parameter. The procedure can modify the value of this parameter, and the modified value is passed back to the calling program. The initial value passed by the caller is ignored.
        - **Example:** `OUT total_salary_out DECIMAL(10,2)`
    - `INOUT` **Parameter:**
        - **Purpose:** Both an input and output parameter. The calling program passes an initial value to the procedure. The procedure can use and modify this value, and the modified value is passed back to the calling program.
        - **Example:** `INOUT counter INT`

- **Syntax for Stored Procedures (MySQL):**

    - Often requires changing the `DELIMITER` temporarily because procedures contain semicolons within their body.
    - `DELIMITER //` (Change delimiter)
    - `CREATE PROCEDURE procedure_name ([parameter_list])`
    - `BEGIN`
    - `-- SQL statements`
    - `END //`
    - `DELIMITER ;` (Restore delimiter)
    - **Calling a Procedure:** `CALL procedure_name([argument_list]);`

Assignment – Lab (Session 13): Stored Procedures

Objective: Develop and execute basic stored procedures with and without parameters, demonstrating their utility.

Commands to Practice:

- **Creating a procedure without parameters:**
    - Create a procedure `GetAllEmployees` that simply selects all records from the `Employees` table.
    - Call the procedure.
    - Create a procedure `CountDepartments` that returns the total count of departments.

- **Creating Procedure with `IN` Parameters:**
  - Create a procedure `GetEmployeeById(IN p_emp_id INT)` that selects employee details for a given `emp_id`.
  - Call the procedure with different `emp_id` values.
  - Create a procedure `GetEmployeesInDept(IN p_dept_id INT)` that lists all employees in a specified department.
- **Creating Procedure with `OUT` Parameters:**
  - Create a procedure `GetAvgSalaryByDept(IN p_dept_id INT, OUT p_avg_salary DECIMAL(10,2))` that calculates the average salary for a given department and returns it via an `OUT` parameter.
  - Call the procedure and display the output parameter.
    - `CALL GetAvgSalaryByDept(10, @avg_sal);`
    - `SELECT @avg_sal;`
- **Creating Procedure with `INOUT` Parameters:**
  - Create a procedure `IncrementSalary(INOUT p_emp_id INT, IN p_percentage DECIMAL(5,2))` that takes an employee ID and a percentage, increases that employee's salary by the percentage, and returns the *new* salary (or a status code) via the `INOUT` parameter. (Or simpler: A procedure that takes a number, increments it, and returns the incremented number).
    - `SET @my_num = 10;`
    - `CALL IncrementNumber(@my_num);`
    - `SELECT @my_num;`

Scenario Example (using `my_company_db`):

1. **No Parameters:**
   - Create a procedure `ListHighSalaryEmployees` that selects `emp_id`, `first_name`, `salary` for employees earning `> 80000`.
2. **`IN` Parameter:**
   - Create `GetDeptLocation(IN p_dept_name VARCHAR(100))` that returns the `location` for a given `dept_name`.
3. **`OUT` Parameter:**
   - Create `GetTotalEmployees(OUT p_total_count INT)` that returns the total number of employees in the `Employees` table.
4. **`INOUT` Parameter (Slightly more complex but good practice):**
   - Create a procedure `UpdateEmployeeSalary(IN p_emp_id INT, IN p_salary_increase_amount DECIMAL(10,2), INOUT p_new_salary DECIMAL(10,2))`. This procedure updates the employee's salary by `p_salary_increase_amount` and sets `p_new_salary` to the employee's updated salary.
     - Inside the procedure:

```
UPDATE Employees
SET salary = salary + p_salary_increase_amount
WHERE emp_id = p_emp_id;

SELECT salary INTO p_new_salary
FROM Employees
WHERE emp_id = p_emp_id;
```

Session 14: Introduction to NoSQL

This session introduces the concept of NoSQL databases, their purpose, different types, and how they differ from traditional RDBMS.

- **Introduction to NoSQL**
  - **Concept:** NoSQL, standing for "Not Only SQL" or "Non-relational SQL", refers to a class of database management systems that do not adhere to the traditional relational database model. They were developed to handle large volumes of unstructured, semi-structured, and polyglot data that are difficult to manage with traditional relational databases, especially in distributed and highly scalable environments.
  - **Emergence:** Gained popularity with the rise of web 2.0, social media, and big data, which demanded high scalability, availability, and flexibility beyond what traditional RDBMS could easily provide.
- **NoSQL Database Types**
  - NoSQL databases are often categorized by their data model:
    - **Key-Value Stores:**
      - **Description:** Simplest NoSQL model. Data is stored as a collection of key-value pairs. Values can be anything from simple strings to complex objects.
      - **Use Cases:** Caching, session management, user profiles, shopping carts.
      - **Examples:** Redis, Amazon DynamoDB, Riak.
    - **Document Databases:**
      - **Description:** Store data in flexible, semi-structured "documents" (usually JSON, BSON, or XML format). Documents can contain nested structures and arrays. Each document can have a different structure, providing schema flexibility.
      - **Use Cases:** Content management systems, e-commerce product catalogs, blogging platforms, mobile applications.
      - **Examples:** MongoDB, Couchbase, Apache Cassandra (can store JSON).
    - **Column-Family Stores (Wide-Column Stores):**
      - **Description:** Data is stored in column families, which are groups of related columns. Unlike traditional row-oriented tables, columns can be added dynamically to individual rows within a column family. Optimized for writes and for reading entire columns.
      - **Use Cases:** Big data analytics, time-series data, large-scale event logging, distributed databases.
      - **Examples:** Apache Cassandra, HBase, Google Bigtable.

        - **Graph Databases:**
            - **Description:** Store data as nodes (entities) and edges (relationships between entities). Optimized for traversing complex relationships quickly.
            - **Use Cases:** Social networks, recommendation engines, fraud detection, knowledge graphs.
            - **Examples:** Neo4j, ArangoDB, Amazon Neptune.

- **Features of NoSQL Database**

    - **Flexible Schema (Schema-less):** No rigid, predefined schema. Data structure can evolve dynamically, making it suitable for rapidly changing data requirements.
    - **Horizontal Scalability:** Designed to scale out by adding more commodity servers (sharding/partitioning) rather than scaling up by adding more powerful hardware.
    - **High Availability:** Often built for distributed environments, providing fault tolerance and high uptime through replication.
    - **Eventual Consistency:** In a distributed system, updates may not be immediately visible across all nodes. Data eventually becomes consistent, balancing availability and performance over strict consistency (part of BASE properties).
    - **BASE Properties (vs. ACID):**
        - **B**asically **A**vailable: The system is guaranteed to be available for reads and writes.
        - **S**oft state: The state of the system may change over time, even without input, due to eventual consistency.
        - **E**ventual consistency: Given enough time, data will propagate to all nodes and become consistent.
    - **Optimized for Specific Workloads:** Each NoSQL type is specialized for certain data models and access patterns, making them highly performant for their specific use cases.
    - **Less Complex Object-Relational Mapping (ORM):** For document databases, mapping application objects to database documents is often more natural than mapping to relational tables.

- **Structured vs. Semi-structured and Unstructured Data**

    - **Structured Data:**
        - **Definition:** Highly organized data that conforms to a fixed schema. It is typically stored in tabular format with rows and columns, where each column has a predefined type and meaning.
        - **Examples:** Relational databases, spreadsheets (CSV, Excel).
        - **Suitability:** RDBMS.
    - **Semi-structured Data:**
        - **Definition:** Data that does not conform to a fixed schema but contains tags or markers that organize elements within the data, creating a hierarchical structure. The structure can vary between different data records.
        - **Examples:** JSON (JavaScript Object Notation), XML (Extensible Markup Language), Avro, Parquet.
        - **Suitability:** Document databases, often processed in big data analytics.
    - **Unstructured Data:**
        - **Definition:** Data that has no predefined structure or organization. It cannot be easily stored in traditional row-and-column databases.

        - **Examples:** Text documents (emails, articles), images, audio files, video files, social media posts.
        - **Suitability:** Specialized databases, data lakes, content management systems, often requires machine learning/AI for analysis.

- **Difference between RDBMS and NoSQL databases**

Feature	RDBMS (Relational Database)	NoSQL Database
Data Model	Tabular (tables, rows, columns), predefined schema.	Key-value, document, column-family, graph; flexible/dynamic schema.
Schema	**Schema-on-write:** Strict, fixed schema defined before data.	**Schema-on-read:** Flexible schema, data can evolve without altering structure.
Scalability	Primarily **Vertical Scalability** (scale up by stronger hardware). Difficult for horizontal.	Primarily **Horizontal Scalability** (scale out by adding more servers).
ACID vs. BASE	Follows **ACID** properties (Atomicity, Consistency, Isolation, Durability) for strong consistency.	Often follows **BASE** properties (Basically Available, Soft state, Eventual consistency) for high availability and performance.
Data Integrity	Strong consistency, referential integrity (foreign keys).	Weaker consistency models, application often handles integrity.
Query Language	**SQL** (Structured Query Language) - declarative.	Varies by type (e.g., MongoDB Query Language, CQL for Cassandra, Gremlin for Neo4j); often programmatic APIs.
Relationships	Joins, primary/foreign keys to define relationships.	Less emphasis on joins; denormalization or embedded documents/relationships.
Typical Use Cases	Financial systems, traditional business apps (ERP, CRM) needing complex transactions.	Big data, real-time web apps, mobile apps, social media, content management, IoT, large-scale distributed systems.
Data Structure	Structured data.	Semi-structured, unstructured, structured data.

Session 15: Introduction to MongoDB

This session provides a practical introduction to MongoDB, a popular NoSQL document database, covering its core concepts, interface, and basic operations.

- **Introduction to MongoDB & Its Benefits**
    - **Concept:** MongoDB is a popular open-source, cross-platform document-oriented database. It falls under the NoSQL category, storing data in flexible, JSON-like documents.
    - **Benefits:**
        - **Flexible Schema:** Allows documents within a collection to have different structures, making it highly adaptable to changing data requirements.

        - **High Performance:** Optimized for high read/write throughput and efficient storage of large volumes of data.
        - **High Availability:** Supports replication (replica sets) for data redundancy and automatic failover.
        - **Horizontal Scalability:** Supports sharding, allowing data to be distributed across multiple servers for massive scalability.
        - **Rich Query Language:** Offers a powerful and expressive query language (MongoDB Query Language - MQL) for complex queries.
        - **Native Object Mapping:** Documents map naturally to objects in most programming languages, simplifying application development.

- **Features of MongoDB**

    - **Document Model:** Stores data in BSON (Binary JSON) documents, which are flexible and hierarchical.
    - **Collections:** Documents are organized into collections, which are analogous to tables in RDBMS, but do not enforce a fixed schema.
    - **Embedded Documents & Arrays:** Supports nesting documents and arrays within other documents, allowing for complex, rich data structures within a single record.
    - **Indexing:** Supports various types of indexes (single field, compound, multi-key, geospatial, text) for fast query performance.
    - **Replication (Replica Sets):** Provides high availability and data redundancy by maintaining multiple copies of data on different servers.
    - **Sharding:** Enables horizontal scaling by distributing data across multiple machines (shards).
    - **Aggregation Framework:** A powerful pipeline-based framework for data aggregation, processing, and transformation.
    - **Ad-hoc Queries:** Supports rich, ad-hoc queries on documents.

- **MongoDB command interface and MongoDB Compass**

    - **MongoDB Command Interface (`mongo` or `mongosh` shell):**
        - **Purpose:** The primary command-line tool for interacting with a MongoDB instance. It's a JavaScript shell that allows you to perform CRUD operations, administer the database, and execute JavaScript code.
        - **Basic Commands:**
            - `mongosh`: Connect to the default MongoDB instance (localhost:27017).
            - `show dbs;`: List all databases.
            - `use database_name;`: Switch to (or create) a database.
            - `db.collection_name.help();`: Get help for collection methods.
            - `db.stats();`: Show database statistics.
    - **MongoDB Compass:**
        - **Purpose:** A free, interactive GUI tool for MongoDB. It allows users to visually analyze, query, and manage their MongoDB data.
        - **Features:** Schema visualization, CRUD operations, aggregation pipeline builder, performance monitoring, index management.
        - **Benefit:** Provides an intuitive way to explore and interact with MongoDB for users who prefer a graphical interface.

- **MongoDB Documents & Collections**
  - **Document:**
    - **Concept:** The basic unit of data in MongoDB. It's a BSON (Binary JSON) document, which is a binary representation of JSON.
    - **Characteristics:** Flexible, self-describing (contains field names and values), supports nested structures and arrays.
    - `_id` **Field:** Every document automatically has a unique `_id` field, which acts as the primary key for the document within a collection.
  - **Collection:**
    - **Concept:** A group of MongoDB documents. It's analogous to a table in a relational database but does not enforce a rigid schema. Documents within a single collection can have different fields.
    - **Dynamic Schema:** MongoDB is schema-less. You can insert documents with varying structures into the same collection without prior definition.

- **Mongo CRUD Operations**
  - **CRUD:** Acronym for Create, Read, Update, Delete – the four basic functions of persistent storage.
  - **Create (Insert):** Add new documents to a collection.
    - `db.collection.insertOne({ field1: "value1", field2: "value2" });`
    - `db.collection.insertMany([{ doc1 }, { doc2 }]);`
  - **Read (Query/Find):** Retrieve documents from a collection.
    - `db.collection.find({});` (Find all documents)
    - `db.collection.find({ field: "value" });` (Find documents matching a condition)
    - `db.collection.find({ field: { $gt: 10 } });` (Using query operators)
    - `db.collection.findOne({ _id: ObjectId("someId") });` (Find a single document by ID)
    - `db.collection.find({}).project({ field1: 1, _id: 0 });` (Select specific fields)
    - `db.collection.find({}).sort({ field: 1 }).limit(5);` (Sort and limit results)
  - **Update:** Modify existing documents in a collection.
    - `db.collection.updateOne({ filter: value }, { $set: { newField: "newValue" } });` (Update a single document)
    - `db.collection.updateMany({ filter: value }, { $inc: { numericField: 5 } });` (Update multiple documents)
    - `$set`: Replaces the value of a field.
    - `$inc`: Increments/decrements a numeric field.
    - `$unset`: Removes a field.
  - **Delete:** Remove documents from a collection.
    - `db.collection.deleteOne({ filter: value });` (Delete a single document)
    - `db.collection.deleteMany({ filter: value });` (Delete multiple documents)
    - `db.collection.deleteMany({});` (Delete all documents in a collection)

- **Aggregation pipeline**
  - **Concept:** MongoDB's powerful framework for performing advanced data aggregation operations. It processes documents through a series of stages (like a pipeline), where each stage transforms the documents and passes the results to the next stage.

- **Common Stages:**
  - `$match`: Filters documents (like SQL `WHERE`).
  - `$group`: Groups documents by a specified key and performs aggregate operations (like SQL `GROUP BY` and aggregate functions).
  - `$project`: Reshapes documents, selects specific fields, renames fields, or adds new computed fields (like SQL `SELECT` list).
  - `$sort`: Sorts documents.
  - `$limit`: Limits the number of documents.
  - `$unwind`: Deconstructs an array field from the input documents to output a document for each element.
- **Example:**

```
db.employees.aggregate([
  { $match: { salary: { $gt: 50000 } } },
  {
    $group: {
      _id: "$department",
      avgSalary: { $avg: "$salary" },
      count: { $sum: 1 },
    },
  },
  { $sort: { avgSalary: -1 } },
  {
    $project: {
      department: "$_id",
      _id: 0,
      averageSalary: "$avgSalary",
      employeeCount: "$count",
    },
  },
]);
```

(This pipeline filters employees by salary, groups them by department to calculate average salary and count, sorts by average salary, and then reshapes the output.)

- **Data models (Embedded vs. Referenced)**
  - **Concept:** How to represent relationships between different pieces of data in a document database. Unlike RDBMS where relationships are primarily established via foreign keys, MongoDB offers two main approaches:
  - **Embedded Data Model:**
    - **Description:** Stores related data in a single document (nesting documents or arrays).
    - **Advantages:**
      - Better performance for read operations (single query retrieves all related data).
      - Atomic operations on a single document are easy.
      - Fewer joins (no joins needed within MongoDB queries).
    - **Disadvantages:**

        - Can lead to large documents (potential for exceeding document size limit).
        - Redundancy if embedded data is duplicated across multiple documents.
        - Updates to embedded data require updating the parent document.
    - **Use Cases:** One-to-few relationships, parent-child relationships where the child cannot exist without the parent (e.g., order and order items, blog post and comments).
    - **Example:**

```
// Blog Post document with embedded comments
{
  _id: ObjectId("post1"),
  title: "My Blog Post",
  author: "John Doe",
  comments: [
    { user: "Alice", text: "Great post!", date: ISODate("2023-01-01") },
    { user: "Bob", text: "Nice insights.", date: ISODate("2023-01-02") }
  ]
}
```

- **Referenced Data Model:**
    - **Description:** Stores related data in separate documents and links them using references (typically the `_id` of the referenced document). Analogous to foreign keys in RDBMS.
    - **Advantages:**
        - Reduces data redundancy.
        - More flexible for complex relationships (many-to-many, highly dynamic relationships).
        - Smaller documents.
    - **Disadvantages:**
        - Requires multiple queries or application-level joins to retrieve related data (more trips to the database).
    - **Use Cases:** One-to-many relationships where the "many" side can be very large, or when the related data needs to be accessed independently.
    - **Example:**

```
// Authors Collection
{ _id: ObjectId("author1"), name: "John Doe" }

// Books Collection (referencing author)
{ _id: ObjectId("book1"), title: "MongoDB Guide", author_id: ObjectId("author1") }
```

        - **Choosing a Model:** Depends on application requirements, query patterns (read-heavy vs. write-heavy), data size, and relationship types. A hybrid approach often works best, combining embedding and referencing.
