**DBMS and RDBMS**

**1. Introduction to Database Management Systems (DBMS):**

**- \*\*Definition and Purpose:**

A Database Management System (DBMS) is software that facilitates the management of databases, serving as an interface between the database and the end-users or application programs. Its purpose is to organize, store, retrieve, and manage data efficiently.

**- \*\*Importance in Information Management:**

DBMS plays a crucial role in information management by providing a structured and efficient way to store and retrieve data. It ensures data integrity, security, and facilitates easy access for authorized users.

**- \*\*Key Components: Data, Database, Database Management System**

- \*\*Data: Raw facts and figures.

- \*\*Database: Organized collection of related data.

- \*\*Database Management System (DBMS):\*\* Software for managing databases.

**2. Relational Database Management System (RDBMS):**

- RDBMS is a type of DBMS that stores data in tables with rows and columns. It enforces a relational model and supports relationships between tables.

**- \*\*Tables, Rows, and Columns in RDBMS:**

- \*\*Tables: Represent entities.

- \*\*Rows: Records or instances.

- \*\*Columns: Attributes or fields.

**- \*\*Primary Keys and Foreign Keys:**

- \*\*Primary Keys: Unique identifiers for rows.

- \*\*Foreign Keys: Establish relationships between tables.

**3. Installation of MySQL Workbench and Local Instance Connection:**

- \*\*Download and Install MySQL Workbench:

Obtain MySQL Workbench from the official website and follow installation instructions for your operating system.

**- \*\*Setup a Local MySQL Server:**

Download and install MySQL Server, configuring it with a root password.

- \*\*Connect MySQL Workbench to Local Server

Create a connection in MySQL Workbench, providing necessary details (hostname, port, username, and password) to connect to the local MySQL Server.

**4. Additional Points on DBMS:**

**- \*\*Data Independence:**

Ability to modify the schema without affecting applications.

**- \*\*Data Definition Language (DDL) and Data Manipulation Language (DML):\*\***

- \*\*DDL: Used for defining database structure.

- \*\*DML: Used for managing data in the database.

**- \*\*ACID Properties:**

ACID properties are a set of characteristics that guarantee the reliability of transactions in a database management system (DBMS). These properties ensure that database transactions are processed reliably, even in the face of failures or errors. The acronym "ACID" stands for Atomicity, Consistency, Isolation, and Durability.

**1. \*\*Atomicity:**

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

- \*Example:\* Consider a fund transfer between two bank accounts. Atomicity ensures that if the debit from one account succeeds, the corresponding credit to the other account will also be executed. If any part of the transaction fails, the entire transaction is rolled back.

**2. \*\*Consistency:**

- \*Definition:\* Consistency ensures that a transaction brings the database from one valid state to another. It preserves the integrity of the database and enforces predefined rules and constraints.

- \*Example:\* In a database where an account balance must never go below zero, consistency ensures that a transaction attempting to deduct more funds than available is rejected, maintaining the integrity of the account balances.

**3. \*\*Isolation:**

- \*Definition:\* Isolation ensures that the concurrent execution of transactions does not result in interference or conflicts. Each transaction appears to be executed in isolation from other transactions.

- \*Example:\* If two transactions are being executed concurrently, isolation ensures that the changes made by one transaction are not visible to the other until the first transaction is committed. This prevents scenarios like "dirty reads" or "uncommitted data."

**4. \*\*Durability:**

- \*Definition:\* Durability ensures that once a transaction is committed, its changes are permanent and survive any subsequent failures, such as power outages or system crashes.

- \*Example:\* If a user makes changes to a database and the system acknowledges the successful commit, durability guarantees that even if the system crashes immediately after, the changes will still be present when the system recovers.

These ACID properties collectively provide a robust framework for ensuring the accuracy, reliability, and integrity of database transactions, making them a fundamental aspect of transactional database systems.

- **\*\*Database Models:**

- Hierarchical, Network, Relational models.

**5. Advantages of DBMS:**

**- \*\*Data Integrity and Security:**

Ensures accuracy and security of data.

**- \*\*Data Consistency:**

Maintains uniformity across the database.

**- \*\*Improved Data Access and Retrieval:**

Facilitates efficient querying and retrieval of data.

**- \*\*Data Independence:**

Allows changes in the database structure without affecting applications.

**- \*\*Concurrent Access and Multi-User Support:**

Supports multiple users accessing the database simultaneously.

**6. Disadvantages of DBMS:**

**- \*\*Cost of Implementation and Maintenance:**

Can be expensive to set up and maintain.

**- \*\*Complexity:**

Learning curve for complex systems.

**- \*\*Security Concerns:**

Requires robust security measures.

**- \*\*Lack of Customization:**

Limited flexibility for customization.

**7. Examples of DBMS and RDBMS:**

- \*\*MySQL, Oracle, Microsoft SQL Server:

Popular examples of DBMS and RDBMS used in various applications.

**8. Introduction to Normalization:**

- \*\*Definition and Purpose:

Normalization is the process of organizing data in a database to reduce redundancy and improve data integrity.

- \*\*Types of Anomalies:

- \*\*Insertion Anomaly: Difficulty in adding data.

- \*\*Update Anomaly: Inconsistency when updating data.

- \*\*Deletion Anomaly: Unintended loss of data.

- \*\*Normal Forms (1NF, 2NF, 3NF, BCNF):\*\*

Gradual stages of normalization to achieve a well-structured database.

**9. Normalization Techniques:**

- \*\*First Normal Form (1NF):

Ensures atomicity of data.

- \*\*Second Normal Form (2NF):

Removes partial dependencies.

- \*\*Third Normal Form (3NF):

Eliminates transitive dependencies.

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

Further refinement to eliminate certain anomalies.

# Difference between DBMS and RDBMS

| **DBMS** | **RDBMS** |
| --- | --- |
| DBMS stores data as file. | RDBMS stores data in tabular form. |
| Data elements need to access individually. | Multiple data elements can be accessed at the same time. |
| No relationship between data. | Data is stored in the form of tables which are related to each other. |
| Normalization is not present. | Normalization is present. |
| DBMS does not support distributed database. | RDBMS supports distributed database. |
| It stores data in either a navigational or hierarchical form. | It uses a tabular structure where the headers are the column names, and the rows contain corresponding values. |
| It deals with small quantity of data. | It deals with large amount of data. |
| Data redundancy is common in this model. | Keys and indexes do not allow Data redundancy. |
| It is used for small organization and deal with small data. | It is used to handle large amount of data. |
| Not all Codd rules are satisfied. | All 12 Codd rules are satisfied. |
| Security is less | More security measures provided. |
| It supports single user. | It supports multiple users. |
| Data fetching is slower for the large amount of data. | Data fetching is fast because of relational approach. |
| The data in a DBMS is subject to low security levels with regards to data manipulation. | There exists multiple levels of data security in a RDBMS. |
| Low software and hardware necessities. | Higher software and hardware necessities. |
| Examples: XML, Window Registry, Forxpro, dbaseIIIplus etc. | Examples: MySQL, PostgreSQL, SQL Server, Oracle, Microsoft Access etc. |

**10. References:**

[**https://www.w3schools.com/sql/default.asp**](https://www.w3schools.com/sql/default.asp)

[**https://www.geeksforgeeks.org/introduction-of-dbms-database-management-system-set-1/?ref=gcse**](https://www.geeksforgeeks.org/introduction-of-dbms-database-management-system-set-1/?ref=gcse)

**Normalization**

1. **What is normalization in the context of database design?**

* **Normalization**
  + **Normalization** is the systematic approach of **reducing redundancy** and **improving data integrity** in a database.
  + It involves breaking down large tables into smaller, well-structured relations.
  + The goal is to create a **clean, efficient, and maintainable** database schema.
* **Why Do We Need Normalization?**
  + **Data anomalies** can occur due to redundancy, leading to issues like insertion, deletion, and update anomalies.
  + Normalization helps eliminate these anomalies, ensuring data consistency and integrity as the database grows.
* **Types of Normal Forms:**
  + **First Normal Form (1NF)**:
    - A relation is in 1NF if it contains **atomic values** (no repeating groups or arrays).
  + **Second Normal Form (2NF)**:
    - A relation is in 2NF if it is in 1NF and all non-key attributes are **fully functionally dependent** on the primary key.
  + **Third Normal Form (3NF)**:
    - A relation is in 3NF if it is in 2NF and **no transitive dependency** exists.
  + **Boyce-Codd Normal Form (BCNF)**:
    - A stronger definition of 3NF, ensuring that every determinant is a **super key**.
  + **Fourth Normal Form (4NF)**:
    - A relation is in 4NF if it is in BCNF and has **no multi-valued dependency**.
  + **Fifth Normal Form (5NF)**:
    - A relation is in 5NF if it is in 4NF and does not contain any **join dependency** (joining should be lossless).
* **Advantages of Normalization:**
  + Minimizes **data redundancy**.
  + Provides a **well-organized** database structure.
  + Ensures **data consistency**.
  + Allows for **flexible database design**.
  + Enforces **relational integrity**.

1. **Why is normalization important for database management?**

**Normalization** is **crucial** for effective **database management**.

* **Minimizing Redundancy**:
  + Normalization helps **reduce data redundancy** by organizing information into smaller, well-structured tables.
  + When data is stored efficiently, we avoid storing the same information multiple times.
  + This minimizes storage space requirements and ensures consistency.
* **Data Integrity and Consistency**:
  + By adhering to normalization rules, we prevent **data anomalies** (such as insertion, deletion, and update issues).
  + Ensuring that data is consistent and accurate is essential for reliable database management.
* **Improved Query Performance**:
  + Well-normalized databases lead to **efficient query execution**.
  + Smaller tables with fewer attributes allow for faster data retrieval.
  + Indexing and searching become more effective.
* **Flexible Schema Design**:
  + Normalization allows for a **flexible schema**.
  + As business requirements change, we can modify the database structure without major disruptions.
  + Adding or removing attributes becomes easier.
* **Relational Integrity**:
  + Normalization enforces **relational integrity constraints**.
  + These constraints maintain the relationships between tables, ensuring data consistency.
  + Referential integrity (foreign keys) is an example of this.
* **Avoiding Anomalies**:
  + **Insertion Anomalies**: Without normalization, adding new data can be problematic due to incomplete information.
  + **Deletion Anomalies**: Removing data may unintentionally delete related information.
  + **Update Anomalies**: Inconsistencies arise when updating data in non-normalized tables.

1. **Explain the concept of data redundancy and how normalization helps to mitigate it.**

* **Data Redundancy:**
  + **Data redundancy** refers to the **repetition of data** in different parts of a database.
  + It occurs when the **same information is stored multiple times** within the database.
  + Redundant data can lead to several problems, including increased storage costs, maintenance challenges, and data inconsistencies.
  + Imagine having identical copies of data scattered across various tables—this is where redundancy creeps in.
* **Problems Caused by Data Redundancy:**
  + Let’s consider an example with a simplified student table:

student\_id | student\_name | student\_age | dept\_id | dept\_name | dept\_head

-----------|--------------|-------------|---------|--------------|-----------

1 | Shiva | 19 | 104 | IT | Jaspreet

2 | Khushi | 18 | 102 | Electronics | Avni Singh

3 | Harsh | 19 | 104 | IT | Jaspreet

* + - Here, we see that some data (e.g., department details) is repeated, causing redundancy.
    - The problems arising from redundancy include:
      1. **Insertion Anomaly**: When adding data, you may need related information (e.g., department details) to avoid incomplete records.
      2. **Deletion Anomaly**: Deleting unrelated data (e.g., a student) might unintentionally remove other relevant data (e.g., department details).
      3. **Updating Anomaly**: Partially updating data (e.g., changing a department head) can lead to inconsistencies.
* **How Normalization Helps:**
  + **Normalization** is a database design technique that aims to **reduce or eliminate redundancy**.
  + It achieves this by breaking down large tables into smaller, well-structured relations.
  + Here’s how normalization mitigates redundancy:
    - **First Normal Form (1NF)**: Ensures atomic values (no repeating groups).
    - **Second Normal Form (2NF)**: Non-key attributes depend fully on the primary key.
    - **Third Normal Form (3NF)**: Eliminates transitive dependencies.
    - **Boyce-Codd Normal Form (BCNF)**: Ensures every determinant is a superkey.
    - **Fourth Normal Form (4NF)**: Addresses multi-valued dependencies.
    - **Fifth Normal Form (5NF)**: Eliminates join dependencies.
  + By adhering to these rules, we create a **clean, efficient, and maintainable** database structure.

1. **What are the primary goals of normalization?**

* **Minimizing Redundancy**:
  + **Normalization** aims to **reduce data redundancy** within a database.
  + By organizing data efficiently, we avoid storing the same information multiple times.
  + This leads to **efficient storage** and ensures **consistency**.
* **Data Integrity and Consistency**:
  + Normalization helps maintain **data integrity** by eliminating anomalies (insertion, deletion, and update issues).
  + Ensuring consistent and accurate data is essential for reliable database management.
* **Efficient Query Performance**:
  + Well-normalized databases result in **faster query execution**.
  + Smaller tables with fewer attributes allow for efficient data retrieval.
  + Indexing and searching become more effective.
* **Flexible Schema Design**:
  + Normalization enables a **flexible schema**.
  + As business requirements change, we can modify the database structure without major disruptions.
  + Adding or removing attributes becomes easier.
* **Relational Integrity**:
  + Normalization enforces **relational integrity constraints**.
  + These constraints maintain relationships between tables, ensuring data consistency.
  + For example, referential integrity (using foreign keys) is part of this process.
* **Avoiding Anomalies**:
  + **Insertion Anomalies**: Normalization prevents incomplete records during data insertion.
  + **Deletion Anomalies**: Ensures that deleting unrelated data doesn’t unintentionally remove relevant information.
  + **Update Anomalies**: Inconsistencies are avoided when updating data.

1. **List and explain the different normal forms in normalization theory.**

* **First Normal Form (1NF)**:
  + A relation is in 1NF if it contains **atomic values** (no repeating groups or arrays).
  + Each attribute holds a single value.
  + No multi-valued or composite attributes.
  + Ensures data is well-structured.
* **Second Normal Form (2NF)**:
  + A relation is in 2NF if it is in 1NF and all non-key attributes are **fully functionally dependent** on the primary key.
  + Non-key attributes should depend solely on the entire primary key, not just part of it.
  + Eliminates partial dependencies.
* **Third Normal Form (3NF)**:
  + A relation is in 3NF if it is in 2NF and **no transitive dependency** exists.
  + Transitive dependency occurs when an attribute depends on another non-key attribute.
  + 3NF ensures data integrity and minimizes redundancy.
* **Boyce-Codd Normal Form (BCNF)**:
  + A stronger definition of 3NF.
  + Every determinant (set of attributes that uniquely determines other attributes) must be a **superkey**.
  + BCNF eliminates anomalies related to functional dependencies.
* **Fourth Normal Form (4NF)**:
  + A relation is in 4NF if it is in BCNF and has **no multi-valued dependency**.
  + Multi-valued dependencies occur when an attribute depends on a subset of the primary key.
  + 4NF further reduces redundancy.
* **Fifth Normal Form (5NF)**:
  + A relation is in 5NF if it is in 4NF and does not contain any **join dependency** (joining should be lossless).
  + Join dependencies involve combining multiple relations without losing information.
  + 5NF ensures the highest level of normalization.

**Advantages of Normalization**:

* Minimizes **data redundancy**.
* Provides a **well-organized** database structure.
* Ensures **data consistency**.
* Allows for **flexible database design**.
* Enforces **relational integrity**.

1. **What is First Normal Form (1NF) and why is it necessary? Explain with example.**

* **What Is First Normal Form (1NF)?**
  + A relation (table) is in 1NF if it satisfies the following conditions:
    - Every attribute (column) in the relation contains only **single-valued** data (no repeating groups or arrays).
    - The **attribute domain** (type of data) remains consistent for each column.
    - Each attribute has a **unique name** within the relation.
    - The order in which data is stored does not matter.
* **Why Is 1NF Necessary?**
  + **Data Integrity**: 1NF ensures that data is **structured correctly**, preventing anomalies during data manipulation.
  + **Redundancy Elimination**: By adhering to 1NF, we minimize **data redundancy**.
  + **Query Efficiency**: Well-organized data allows for efficient querying and retrieval.
* **Example of 1NF**:
  + Let’s consider a simplified example with a relation called **STUDENT**:
  + STUDENT (ID, Name, Courses)
    - In the initial table, the **Courses** attribute contains multiple values (a multi-valued attribute):
    - ID | Name | Courses
    - -------------------------
    - 1 | A | c1, c2
    - 2 | E | c3
    - 3 | M | c2, c3
    - This violates 1NF because of the multi-valued attribute **Courses**.
    - To achieve 1NF, we decompose it into a new table:
    - ID | Name | Course
    - -----------------------
    - 1 | A | c1
    - 1 | A | c2
    - 2 | E | c3
    - 3 | M | c2
    - 3 | M | c3
    - Now each attribute contains only single-valued data, and the table is in 1NF.
* **Conclusion**:
  + 1NF establishes the foundation for more complex normalization strategies.
  + It ensures that data is organized to facilitate data processing, remove redundancy, and support data integrity.

1. **How does Second Normal Form (2NF) differ from First Normal Form (1NF)?explain with example**

* **First Normal Form (1NF)**:
  + **Objective**: Ensure that data is organized into atomic values (no repeating groups or arrays).
  + **Conditions for 1NF**:
    - Each attribute contains only **single-valued** data.
    - No multi-valued or composite attributes.
    - Unique names for each attribute within the table.
  + **Example**: Consider the following relation:
  + Roll Number | Student Name | Marks
  + ----------------------------------
  + 1 | Abhay | 96
  + 2 | Amit | 78
  + 3 | Ayushi | 86

This relation is in 1NF because it contains only single-valued attributes and no multi-valued or composite attributes.

* **Second Normal Form (2NF)**:
  + **Objective**: Address **partial dependency** in relations with composite keys.
  + **Conditions for 2NF**:
    - The relation must be in 1NF.
    - No partial dependency (non-prime attributes depend fully on the entire primary key).
  + **Example**: Let’s consider a functional dependency for the relation **R (X, Y, E, F)**:
    - Functional dependencies: {XY → EF, E → F}
    - The closure of (XY) is {X, Y, E, F}, making XY the candidate key.
    - For each functional dependency:
      1. XY → EF: No partial dependency; non-prime attributes depend on the whole candidate key.
      2. E → F: No partial dependency; non-prime attributes depend on each other only.
    - Thus, the relation satisfies 2NF.
* **Summary of Differences**:
  + **1NF**:
    - Ensures atomicity and absence of composite or multi-valued attributes.
    - Does not necessarily handle functional dependencies.
  + **2NF**:
    - Builds upon 1NF.
    - Eliminates partial dependency.
    - Ensures that non-prime attributes depend fully on the entire candidate key.

1. **Describe Third Normal Form (3NF) and its significance in database design. Explain with example.**

* **Third Normal Form (3NF)**:
  + **Objective**: 3NF aims to eliminate **transitive dependencies** within a relation.
  + A relation is in 3NF if:
    - It is already in **Second Normal Form (2NF)**.
    - No non-prime attribute is transitively dependent on the primary key.
  + In simpler terms, 3NF ensures that non-key attributes depend directly on the primary key, avoiding indirect dependencies.
* **Significance of 3NF**:
  + **Data Integrity**: 3NF improves data integrity by minimizing redundancy and ensuring consistent information.
  + **Effective Organization**: It organizes data efficiently, making queries and updates more straightforward.
  + **Avoiding Anomalies**: 3NF eliminates insertion, deletion, and update anomalies.
* **Example**: Consider a table storing information about **employees** and their **projects**:
* EMPLOYEE\_PROJECTS (EmployeeID, EmployeeName, ProjectID, ProjectName, Manager)
  + In this table, the **Manager** attribute depends on **EmployeeID**, not directly on the primary key.
  + This violates 3NF because **Manager** is transitively dependent on the primary key.
  + To bring it to 3NF, we split the table into two separate tables:

**Employees Table**:

EmployeeID | EmployeeName

-------------------------

1 | Abhay

2 | Amit

3 | Ayushi

**Projects Table**:

ProjectID | ProjectName | Manager

---------------------------------

101 | ProjectA | Abhay

102 | ProjectB | Amit

103 | ProjectC | Ayushi

* + Now, each attribute depends directly on the primary key, and the table is in 3NF.
* **Conclusion**:
  + 3NF ensures that non-key properties only depend on the primary key, removing redundancy and helping create a well-organized and normalized relational database model.

1. **What is Boyce-Codd Normal Form (BCNF) and how does it differ from Third Normal Form (3NF)? explain with example.**.

* **Third Normal Form (3NF)**:
  + **Objective**: 3NF aims to eliminate **transitive dependencies** within a relation.
  + A relation is in 3NF if:
    - It is already in **Second Normal Form (2NF)**.
    - No non-prime attribute is transitively dependent on the primary key.
  + In simpler terms, 3NF ensures that non-key attributes depend directly on the primary key, avoiding indirect dependencies.
* **Boyce-Codd Normal Form (BCNF)**:
  + **Objective**: BCNF stands for Boyce-Codd normal form and was introduced by R.F. Boyce and E.F. Codd in 1974.
  + A functional dependency is said to be in BCNF if these properties hold:
    - It should already be in 3NF.
    - For a functional dependency say **P → Q**, **P** should be a **super key**.
  + BCNF is an extension of 3NF and has stricter rules than 3NF. It is considered stronger than 3NF.
* **Example**:
  + Consider a relation **R (A, B, C, D)** with the following functional dependencies:
  + A → B
  + A → C
  + C → D
  + C → A
  + The candidate keys are: {A, C}.
  + Let’s analyze:
    - **Closure of A**: {A, B, C, D}
    - **Closure of C**: {A, B, C, D}
  + This relation is in BCNF because:
    - It is already in 3NF (no prime attribute derives a non-prime attribute).
    - On the left-hand side of the functional dependency, there is a candidate key (A).
* **Differences between 3NF and BCNF**:
  + **3NF**:
    - Ensures no transitive dependency (no non-prime attribute transitively dependent on the candidate key).
    - Less strict than BCNF.
    - Preserves all functional dependencies.
  + **BCNF**:
    - Requires that for any functional dependency **P → Q**, **P** must be a super key.
    - Stricter than 3NF.
    - May or may not preserve all functional dependencies.
    - More challenging to achieve.

1. **Explain the concept of transitive dependency and its role in normalization.**

* **Transitive Dependency**:
  + **Transitive dependency** occurs when an attribute **indirectly depends** on another attribute through a third attribute.
  + In simpler terms, if **A → B** and **B → C**, then **A → C** is a transitive dependency.
  + It introduces redundancy and can lead to data anomalies.
* **Role in Normalization**:
  + **Normalization** aims to organize data efficiently, eliminate redundancy, and ensure data integrity.
  + Transitive dependencies violate these goals.
  + By removing transitive dependencies, we achieve higher levels of normalization.
* **Example**: Consider a relation **STUDENT (StudentID, Course, Professor)**:
  + Functional dependencies:
    - **StudentID → Course**
    - **Course → Professor**
  + Here, **StudentID → Professor** is a transitive dependency.
  + To remove it, we split the relation into two tables:

**Students Table**:

StudentID | Course

-----------------

1 | Math

2 | History

**Courses Table**:

Course | Professor

-----------------

Math | Dr. Smith

History| Prof. Johnson

* **Conclusion**:
  + Transitive dependencies are undesirable in a well-structured database.
  + Normalization helps eliminate them, ensuring efficient data management and consistency.

1. **Can you provide examples illustrating the process of normalization and its application in real-world database scenarios?**

* **Student Database Example**:
  + **Scenario**: Imagine a university managing student records.
  + **Initial Table (Not Normalized)**:
  + StudentID | StudentName | Courses
  + ---------------------------------
  + 1 | John | Math, History
  + 2 | Jane | Chemistry
  + **Issue**: The **Courses** attribute contains multiple values (a multi-valued attribute).
  + **Normalization Solution**:
    - Split the table into two separate tables:
      1. **Students Table**:
      2. StudentID | StudentName
      3. -----------------------
      4. 1 | John
      5. 2 | Jane
      6. **Courses Table**:
      7. StudentID | Course
      8. -----------------
      9. 1 | Math
      10. 1 | History
      11. 2 | Chemistry
* **Employee-Department Example**:
  + **Scenario**: An organization tracks employee information and their associated departments.
  + **Initial Table (Not Normalized)**:
  + EmployeeID | EmployeeName | Department
  + ---------------------------------------
  + 101 | Alice | HR, Finance
  + 102 | Bob | IT
  + **Issue**: The **Department** attribute contains multiple values.
  + **Normalization Solution**:
    - Create separate tables for employees and departments:
      1. **Employees Table**:
      2. EmployeeID | EmployeeName
      3. -------------------------
      4. 101 | Alice
      5. 102 | Bob
      6. **Departments Table**:
      7. EmployeeID | Department
      8. -----------------------
      9. 101 | HR
      10. 101 | Finance
      11. 102 | IT
* **Product Inventory Example**:
  + **Scenario**: A retail store manages product inventory.
  + **Initial Table (Not Normalized)**:
  + ProductID | ProductName | Categories
  + -------------------------------------
  + 1 | Laptop | Electronics, Computers
  + 2 | Shirt | Clothing
  + **Issue**: The **Categories** attribute contains multiple values.
  + **Normalization Solution**:
    - Split the table into two related tables:
      1. **Products Table**:
      2. ProductID | ProductName
      3. -----------------------
      4. 1 | Laptop
      5. 2 | Shirt
      6. **Categories Table**:
      7. ProductID | Category
      8. --------------------
      9. 1 | Electronics
      10. 1 | Computers
      11. 2 | Clothing
* **Benefits of Normalization**:
  + **Efficiency**: Well-organized data allows for faster queries and updates.
  + **Data Integrity**: Reduces anomalies (insertion, deletion, update issues).
  + **Consistency**: Ensures accurate and reliable information.
  + **Flexible Design**: Allows schema modifications without disruptions.

1. **Define SQL constraints and explain their significance in database management. Provide examples of different types of SQL constraints.**

### What Are SQL Constraints?

* **SQL constraints** are rules applied to database tables, ensuring the correctness, consistency, and dependability of stored data.
* These rules define boundaries and prerequisites for the values that can be inserted into a table, thereby preventing the inclusion of inappropriate or invalid data.

### Types of SQL Constraints:

1. **NOT NULL Constraint**:
   * Ensures that a column cannot have a **NULL value**.
   * Example:
   * CREATE TABLE Students (

student\_id INT NOT NULL,

student\_name VARCHAR(50) NOT NULL

);

1. **UNIQUE Constraint**:
   * Ensures that all values in a column are **different** (no duplicates).
   * Example:
   * CREATE TABLE Employees (

employee\_id INT UNIQUE,

employee\_email VARCHAR(100) UNIQUE

);

1. **PRIMARY KEY Constraint**:
   * A combination of **NOT NULL** and **UNIQUE** constraints.
   * Uniquely identifies each row in a table.
   * Example:
   * CREATE TABLE Orders (

order\_id INT PRIMARY KEY,

customer\_id INT NOT NULL

);

1. **FOREIGN KEY Constraint**:
   * References a row in another table.
   * Ensures data consistency and maintains relationships between tables.
   * Example:

* CREATE TABLE OrderDetails (

order\_id INT,

product\_id INT,

FOREIGN KEY (order\_id) REFERENCES Orders(order\_id),

FOREIGN KEY (product\_id) REFERENCES Products(product\_id));

1. **CHECK Constraint**:
   * Validates a condition for new values.
   * Example:
   * CREATE TABLE Employees (

employee\_id INT PRIMARY KEY,

salary DECIMAL CHECK (salary >= 30000));

1. **DEFAULT Constraint**:
   * Sets a default value for a column if no value is specified during insertion.
   * Example:
   * CREATE TABLE Customers (

customer\_id INT PRIMARY KEY, registration\_date DATE DEFAULT CURRENT\_DATE);

### Significance of SQL Constraints:

* **Data Integrity**: Constraints ensure that data adheres to predefined rules, preventing inconsistencies.
* **Consistency**: Constraints maintain relationships and prevent invalid data.
* **Efficiency**: Well-defined constraints lead to efficient queries and updates.
* **Security**: Constraints protect data from unauthorized modifications.

1. **Discuss the purpose of the NOT NULL constraint in SQL. How does it differ from the UNIQUE constraint?**

### NOT NULL Constraint:

* The **NOT NULL** constraint ensures that a column **cannot contain NULL values**.
* In other words, it mandates that every entry in that column must have a valid value.
* It is commonly used for columns that are essential for data integrity or identification.
* For example, when registering at a hospital, the **date of birth** field should not be left empty.

**Example**: Suppose we have an **Employees** table with an **employee\_id** column that serves as the primary key. We want to ensure that every employee has a valid ID:

CREATE TABLE Employees (

employee\_id INT PRIMARY KEY NOT NULL,

employee\_name VARCHAR(50),

department VARCHAR(50)

);

### UNIQUE Constraint:

* The **UNIQUE** constraint ensures that all values in a column are **different** (no duplicates).
* It guarantees uniqueness for a column or a set of columns.
* Unlike the **PRIMARY KEY**, which is also unique, you can have **multiple UNIQUE constraints** per table.

**Example**: Consider an **Emails** table where we want to ensure that each email address is unique:

CREATE TABLE Emails (

email\_id INT PRIMARY KEY,

email\_address VARCHAR(100) UNIQUE,

user\_id INT

);

### Differences:

* **Purpose**:
  + **NOT NULL**: Ensures that a column does not hold NULL values.
  + **UNIQUE**: Ensures that all values in a column are different.
* **Applicability**:
  + **NOT NULL** is applied to columns that require valid data (e.g., primary keys, essential fields).
  + **UNIQUE** is used to prevent duplicate values in a column.
* **Combination with PRIMARY KEY**:
  + A **PRIMARY KEY** automatically has a **UNIQUE constraint**.
  + You can have **multiple UNIQUE constraints** but only one PRIMARY KEY constraint per table.

1. **Explain the concept of a PRIMARY KEY constraint in SQL. What role does it play in database design and data integrity?** **Explain the concept of a PRIMARY KEY constraint in SQL. What role does it play in database design and data integrity?**

### What Is a PRIMARY KEY Constraint?

* A **PRIMARY KEY** constraint is a fundamental concept in SQL.
* It **uniquely identifies each record (row) in a table**.
* Key properties of a PRIMARY KEY:
  + **Uniqueness**: Each value in the primary key column(s) must be unique across all rows.
  + **Non-nullability**: A primary key column cannot contain NULL values.
  + **Single or Composite**: The primary key can consist of a single column or multiple columns (composite key).

### Role of PRIMARY KEY in Database Design:

1. **Uniqueness and Identification**:
   * The primary key ensures that each row in a table has a **unique identifier**.
   * It allows us to distinguish and locate specific records efficiently.
2. **Data Integrity**:
   * By enforcing uniqueness and non-nullability, the primary key maintains **data integrity**.
   * It prevents duplicate or incomplete data.
3. **Relationships and Joins**:
   * The primary key is often used as a reference in **relationships** between tables.
   * It facilitates **joins** between related tables.

### Example:

Suppose we have an **Employees** table with an **employee\_id** column as the primary key. Here’s how it plays a role:

CREATE TABLE Employees (

employee\_id INT PRIMARY KEY,

employee\_name VARCHAR(50),

department VARCHAR(50)

);

* The **employee\_id** uniquely identifies each employee.
* It ensures that no two employees have the same ID.
* Relationships with other tables (e.g., salary, projects) can be established using this primary key.

1. **Explain the difference between Data Definition Language (DDL), Data Manipulation Language (DML), and Data Control Language (DCL) in SQL.Provide examples of scenarios where DDL commands would be used.**

### Data Definition Language (DDL):

1. **Purpose**:
   * DDL deals with **defining and managing the structure of database objects** (e.g., tables, indexes, views, and constraints).
   * It focuses on creating, altering, and deleting database structures.
   * DDL commands are typically executed by database administrators or during initial database setup.
2. **Common DDL Commands**:
   * **CREATE**: Used to create new database objects (e.g., tables, indexes, views).
   * **ALTER**: Modifies the structure of existing database objects (e.g., adding or renaming columns).
   * **DROP**: Deletes database objects (e.g., dropping a table).
3. **Scenarios for DDL Commands**:
   * **Creating Tables**: When setting up a new application, you create tables to store data.
   * **Modifying Schema**: When adding new columns or changing data types.
   * **Managing Indexes**: Creating or altering indexes for efficient data retrieval.
   * **Setting Constraints**: Defining primary keys, unique constraints, and foreign keys.

### Example Scenarios for DDL Commands:

* **Creating a New Table**:
  + Suppose you’re building an e-commerce platform. You’d use DDL to create a **Products** table to store product information:
  + CREATE TABLE Products (
  + product\_id INT PRIMARY KEY,
  + product\_name VARCHAR(100),
  + price DECIMAL(10, 2)
  + );
* **Adding a New Column**:
  + Imagine you want to track product availability. You’d alter the existing **Products** table:
  + ALTER TABLE Products
  + ADD COLUMN stock\_quantity INT;
* **Creating Indexes**:
  + To improve search performance, you’d create an index on the **product\_name** column:
  + CREATE INDEX idx\_product\_name ON Products (product\_name);

**SQL QUERY**

SQL queries are used to retrieve, manipulate, and manage data stored in a relational database. Here are some common types of SQL queries:

**SELECT Query:** Used to retrieve data from one or more tables.

SELECT column1, column2

FROM table\_name

WHERE condition;

**INSERT Query:**The INSERT statement adds new records (rows) into a table.

INSERT INTO table\_name (column1, column2)

VALUES (value1, value2);

**UPDATE Query**: UPDATE statement modifies existing records in a table.

UPDATE table\_name

SET column1 = value1, column2 = value2

WHERE condition;

**DELETE Query:** Used to delete existing records from a table.

DELETE FROM table\_name

WHERE condition;

**Operators in WHERE Clause:**

Operators in the WHERE clause are used to specify conditions for filtering rows returned by a SELECT statement.

**Comparison Operators:** Used to compare values.

=: Equal to

<: Less than

>: Greater than

<=: Less than or equal to

>=: Greater than or equal to

<> or !=: Not equal to

**Logical Operators:** Used to combine multiple conditions.

AND: Returns true if all conditions are true.

OR: Returns true if at least one condition is true.

NOT: Negates a condition.

**IN Operator:** Checks if a value matches any value in a list.

WHERE column\_name IN (value1, value2, ...);

**BETWEEN Operator**: Checks if a value is within a range.

WHERE column\_name BETWEEN value1 AND value2;

**Joins**

1. \*\*INNER JOIN:

- Returns rows when there is a match in both tables based on the specified condition.

- The `ON` keyword is used to specify the condition for the match.

```sql

SELECT \* FROM table1

INNER JOIN table2 ON table1.column = table2.column;

```

2. \*\*LEFT JOIN (or LEFT OUTER JOIN):

- Returns all rows from the left table and the matching rows from the right table.

- If there is no match, NULL values are returned for columns from the right table.

```sql

SELECT \* FROM table1

LEFT JOIN table2 ON table1.column = table2.column;

```

3. \*\*RIGHT JOIN (or RIGHT OUTER JOIN):

- Returns all rows from the right table and the matching rows from the left table.

- If there is no match, NULL values are returned for columns from the left table.

```sql

SELECT \* FROM table1

RIGHT JOIN table2 ON table1.column = table2.column;

```

4. \*\*FULL JOIN (or FULL OUTER JOIN):

- Returns all rows when there is a match in either the left or right table.

- If there is no match, NULL values are returned for columns from the table without a match.

```sql

SELECT \* FROM table1

FULL JOIN table2 ON table1.column = table2.column;

```

5. \*\*CROSS JOIN:

- Returns the Cartesian product of both tables, meaning every row from the left table is combined with every row from the right table.

- No specific condition is used.

```sql

SELECT \* FROM table1

CROSS JOIN table2;

```

6. \*\*SELF JOIN:

- When a table is joined with itself, typically used when dealing with hierarchical data or comparisons within the same table.

```sql

SELECT \* FROM employees e1

INNER JOIN employees e2 ON e1.manager\_id = e2.employee\_id;

```

**Aggregate functions**

Aggregate functions in SQL are used to perform a calculation on a set of values and return a single, summarized result. Here are some commonly used aggregate functions:

1. \*\*COUNT():

- Counts the number of rows in a specified column or the number of rows that meet a certain condition.

```sql

SELECT COUNT(column\_name) FROM table\_name;

```

2. \*\*SUM():

- Calculates the sum of values in a numeric column.

```sql

SELECT SUM(column\_name) FROM table\_name;

```

3. \*\*AVG():

- Calculates the average (mean) of values in a numeric column.

```sql

SELECT AVG(column\_name) FROM table\_name;

```

4. \*\*MIN():

- Retrieves the minimum value in a column.

```sql

SELECT MIN(column\_name) FROM table\_name;

```

5. \*\*MAX():

- Retrieves the maximum value in a column.

```sql

SELECT MAX(column\_name) FROM table\_name;

```

These aggregate functions can be used in combination with the `GROUP BY` clause to perform calculations on groups of rows based on the values in one or more columns. For example:

```sql

SELECT department, AVG(salary) AS avg\_salary

FROM employees

GROUP BY department;

```

In this example, the `AVG()` function is used with the `GROUP BY` clause to calculate the average salary for each department.

Additionally, you can use the `HAVING` clause to filter the results of a `GROUP BY` based on a specified condition. For example:

```sql

SELECT department, AVG(salary) AS avg\_salary

FROM employees

GROUP BY department

HAVING AVG(salary) > 50000;

```

This query retrieves departments with an average salary greater than 50,000.