Independent Skill Development Mission

Independent Skill Development Mission NXT
EMPOWERING FUTURES, SHAPING TOMORROW

ISDM (INDEPENDENT SKILL DEVELOPMENT MISSION

UNDERSTANDING 1NF, 2NF, 3NF, AND BCNF

CHAPTER 1: INTRODUCTION TO DATABASE NORMALIZATION

Definition and Importance of Normalization

Database **normalization** is a process used in relational databases to **reduce data redundancy** and **improve data integrity** by organizing data into structured tables. It ensures that **data anomalies** (insertion, deletion, and update anomalies) are minimized and that the database operates efficiently.

Normalization follows a series of rules known as **Normal Forms (NF)**, which include:

1. **First Normal Form (1NF)** - Eliminates duplicate columns and ensures atomicity.
2. **Second Normal Form (2NF)** - Ensures that data depends on the entire primary key.
3. **Third Normal Form (3NF)** - Eliminates transitive dependencies.
4. **Boyce-Codd Normal Form (BCNF)** - A stronger version of 3NF that handles more complex dependencies.

By following these normalization principles, databases become more scalable, flexible, and easier to maintain.

CHAPTER 2: FIRST NORMAL FORM (1NF) – ELIMINATING DUPLICATE DATA AND ENSURING ATOMICITY

Definition of 1NF

A table is in **First Normal Form (1NF)** if:

- All columns contain **atomic values** (each column holds only one value per row).
- Each row is uniquely identifiable using a **primary key**.
- No **repeating groups** or arrays exist in any column.

Example of a Table Not in 1NF

Student_ID	Name	Courses
101	Alice	Math, Science
102	Bob	English
103	Carol	Math, History

Here, the **Courses** column contains multiple values, violating **atomicity**.

Correcting the Table to 1NF

Student_ID	Name	Course
101	Alice	Math
101	Alice	Science
102	Bob	English
103	Carol	Math

103	Carol	History

Now, each column contains only atomic values, and **repeating groups** are removed.

Exercise

- Identify a dataset in your database that violates **1NF** and restructure it into a **1NF-compliant table**.
- Write an SQL query to separate multi-valued columns into individual rows.

Chapter 3: Second Normal Form (2NF) – Removing Partial Dependencies

Definition of 2NF

A table is in **Second Normal Form (2NF)** if:

- It is in **1NF**.
- **No partial dependency exists** (all non-key attributes must depend on the whole **primary key**, not just part of it).

Example of a Table Not in 2NF

Consider the following table for student course enrollments:

Student_ID	Course_ID	Student_Name	Course_Name
101	MATH101	Alice	Mathematics
101	SCI101	Alice	Science
102	ENG101	Bob	English

Here, **Student_Name** depends only on **Student_ID**, and **Course_Name** depends only on **Course_ID**, creating a **partial dependency**.

Correcting the Table to 2NF

1. **Create a Student Table:**

Student_ID	Student_Name
101	Alice
102	Bob

2. **Create a Course Table:**

Course_ID	Course_Name
MATH101	Mathematics
SCI101	Science
ENG101	English

3. **Create a Student-Course Mapping Table:**

Student_ID	Course_ID
101	MATH101
101	SCI101
102	ENG101

Now, all non-key attributes fully depend on the **primary key**, satisfying **2NF**.

Exercise

- Identify a table that violates **2NF** in your database.
- Split it into multiple tables to eliminate **partial dependencies**.

Chapter 4: Third Normal Form (3NF) – Eliminating Transitive Dependencies

Definition of 3NF

A table is in **Third Normal Form (3NF)** if:

- It is in **2NF**.
- **No transitive dependency exists** (non-key columns must depend only on the **primary key** and not on other non-key columns).

Example of a Table Not in 3NF

Order_ID	Customer_ID	Customer_Name	Customer_Address
501	1001	John Doe	NY, USA
502	1002	Alice Smith	LA, USA

Here, **Customer_Name** and **Customer_Address** depend on **Customer_ID**, not on **Order_ID**, creating a **transitive dependency**.

Correcting the Table to 3NF

1. **Create an Orders Table:**

Order_ID	Customer_ID
501	1001
502	1002

2. **Create a Customers Table:**

Customer_ID	Customer_Name	Customer_Address
1001	John Doe	NY, USA
1002	Alice Smith	LA, USA

Now, non-key columns depend **only** on the primary key, ensuring **3NF compliance**.

Exercise

- Find a table that violates **3NF** in your database.
- Normalize it by **removing transitive dependencies**.

Chapter 5: Boyce-Codd Normal Form (BCNF) – The Highest Level of Normalization

Definition of BCNF

A table is in **BCNF** if:

- It is in **3NF**.
- Every **determinant** is a **candidate key** (if one attribute determines another, it must be a key).

Example of a Table Not in BCNF

Professor_ID	Course_ID	Professor_Name	Department
201	MATH101	Dr. Smith	Math
202	ENG101	Dr. Brown	English
201	SCI101	Dr. Smith	Math

Here, **Professor_Name** determines **Department**, but **Professor_ID** is not a candidate key, violating **BCNF**.

Correcting the Table to BCNF

1. **Create a Professor Table:**

Professor_ID	Professor_Name	Department
201	Dr. Smith	Math
202	Dr. Brown	English

2. **Create a Course Assignment Table:**

Professor_ID	Course_ID
201	MATH101
202	ENG101
201	SCI101

Now, all **determinants** are **candidate keys**, making it **BCNF-compliant**.

Exercise

- Identify a **BCNF violation** in your database.
- Normalize the table to ensure **BCNF compliance**.

Conclusion

Normalization ensures **data integrity, reduces redundancy, and improves query efficiency**. By following **1NF, 2NF, 3NF, and BCNF**, databases become more structured, scalable, and efficient. 🚀

ER Diagrams & Schema Design

Chapter 1: Introduction to ER Diagrams and Schema Design

Definition and Importance

Entity-Relationship (ER) Diagrams and Schema Design are fundamental concepts in **database modeling**. An **ER Diagram (ERD)** visually represents entities (tables), their attributes (columns), and relationships between them. This helps database designers create a **structured, efficient, and scalable** database.

A well-designed **Schema** defines the database structure, ensuring **data integrity, reducing redundancy, and improving performance**. Schema design involves organizing tables, defining relationships, and ensuring normalization principles (1NF, 2NF, 3NF, BCNF) are met.

Using **ER Diagrams** before implementing a database is essential because it:

- Provides a **clear blueprint** for the database structure.
- Helps in identifying **primary keys, foreign keys, and relationships**.
- Reduces **data duplication and inconsistencies**.
- Improves **query performance and scalability**.

Chapter 2: Components of an ER Diagram

1. Entities (Tables in a Database)

Entities represent **real-world objects** in a database. Each entity is stored as a **table** with attributes as columns.

Example:

- **Student (Entity)** with attributes: **Student_ID, Name, Age, Email**.
- **Course (Entity)** with attributes: **Course_ID, Course_Name, Credits**.

Entities are classified as:

- **Strong Entity** – Exists independently (e.g., Student, Course).
- **Weak Entity** – Dependent on another entity (e.g., Order Details in an e-commerce system).

2. Attributes (Columns in a Table)

Each entity has attributes that define its properties. Attributes can be:

- **Primary Key (PK):** Uniquely identifies a record (e.g., Student_ID).
- **Foreign Key (FK):** Establishes relationships between tables (e.g., Course_ID in a Student-Course table).
- **Composite Attribute:** Made up of multiple values (e.g., Full Name = First Name + Last Name).
- **Derived Attribute:** Calculated from other attributes (e.g., Age from Date of Birth).

3. Relationships (Links Between Entities)

Relationships define **how entities interact** in a database. They are represented by **lines** connecting entities in an ERD.

- **One-to-One (1:1):** A student has one ID card.
- **One-to-Many (1:M):** A teacher teaches multiple courses.
- **Many-to-Many (M:N):** A student enrolls in multiple courses, and a course has multiple students.

Exercise

- Identify **entities and attributes** for an online bookstore database.
- Define **primary keys** and **foreign keys** for each entity.
- Sketch an ER diagram using software like **MySQL Workbench, Lucidchart, or Draw.io**.

Chapter 3: Converting ER Diagrams to Schema Design

Step 1: Identifying Entities and Attributes

Start by listing the main entities and their attributes.

Example: School Database

- **Students (Student_ID, Name, Email, Age)**
- **Courses (Course_ID, Course_Name, Credits)**
- **Enrollments (Enrollment_ID, Student_ID, Course_ID, Grade)**

Step 2: Defining Relationships

Using **primary keys and foreign keys**, establish relationships between tables.

Example Schema Design for School Database

```
CREATE TABLE Students (
   Student_ID INT PRIMARY KEY,
   Name VARCHAR(100),
   Email VARCHAR(100) UNIQUE,
   Age INT
);

CREATE TABLE Courses (
   Course_ID INT PRIMARY KEY,
   Course_Name VARCHAR(100),
   Credits INT
);

CREATE TABLE Enrollments (
   Enrollment_ID INT PRIMARY KEY,
   Student_ID INT,
   Course_ID INT,
   Grade VARCHAR(2),
   FOREIGN KEY (Student_ID) REFERENCES Students(Student_ID),
   FOREIGN KEY (Course_ID) REFERENCES Courses(Course_ID)
);
```

Here, the **Enrollments table** connects **Students** and **Courses**, following **1NF, 2NF, and 3NF principles** to prevent redundancy.

Exercise

- Convert an ER diagram into **SQL Schema** for an **E-commerce system** (Products, Orders, Customers).
- Write **SQL queries** to fetch data from multiple tables.

Chapter 4: Normalization in Schema Design

1. Avoiding Data Redundancy

Unnormalized databases may store **repeated data**, leading to inconsistencies.

Example of Unnormalized Table

Order_ID	Customer_Name	Product_Name	Quantity	Price
101	Alice	Laptop	1	800
101	Alice	Mouse	1	20
102	Bob	Keyboard	1	50

Here, **Customer_Name** repeats unnecessarily.

2. Applying Normalization (1NF, 2NF, 3NF)

Breaking the table into **separate entities**:

Customers Table

Customer_ID	Customer_Name
1	Alice

2	Bob

Orders Table

Order_ID	Customer_ID
101	1
102	2

Order_Details Table

Order_ID	Product_Name	Quantity	Price
101	Laptop	1	800
101	Mouse	1	20

Now, **redundancy is minimized**, and database performance improves.

Exercise

- Take an **unnormalized table** and convert it into **3NF**.
- Write a **JOIN query** to retrieve order details for a specific customer.

CHAPTER 5: CASE STUDY – DESIGNING AN ER DIAGRAM FOR AN E-COMMERCE PLATFORM

Scenario

A company wants to design an **E-Commerce Database** that includes:

- Customers placing orders.

- Orders containing multiple products.
- Payments linked to orders.

Step 1: Identifying Entities

1. **Customers (Customer_ID, Name, Email, Address)**
2. **Products (Product_ID, Name, Price, Category)**
3. **Orders (Order_ID, Customer_ID, Order_Date, Status)**
4. **Order_Details (Order_ID, Product_ID, Quantity, Subtotal)**
5. **Payments (Payment_ID, Order_ID, Payment_Method, Payment_Status)**

Step 2: Defining Relationships

- **One-to-Many:** A customer places multiple orders.
- **Many-to-Many:** An order contains multiple products.
- **One-to-One:** An order has a single payment.

Step 3: Creating the ER Diagram

Using **Lucidchart**, **Draw.io**, or **MySQL Workbench**, design the following **ERD**:

(Sample ERD – replace with actual tool output)

Step 4: SQL Schema Design

```
CREATE TABLE Customers (
  Customer_ID INT PRIMARY KEY,
  Name VARCHAR(100),
```

```
    Email VARCHAR(100) UNIQUE,
    Address TEXT
);

CREATE TABLE Products (
    Product_ID INT PRIMARY KEY,
    Name VARCHAR(100),
    Price DECIMAL(10,2),
    Category VARCHAR(50)
);

CREATE TABLE Orders (
    Order_ID INT PRIMARY KEY,
    Customer_ID INT,
    Order_Date DATE,
    Status VARCHAR(20),
    FOREIGN KEY (Customer_ID) REFERENCES
Customers(Customer_ID)
);

CREATE TABLE Order_Details (
```

```
    Order_ID INT,
    Product_ID INT,
    Quantity INT,
    Subtotal DECIMAL(10,2),
    PRIMARY KEY (Order_ID, Product_ID),
    FOREIGN KEY (Order_ID) REFERENCES Orders(Order_ID),
    FOREIGN KEY (Product_ID) REFERENCES Products(Product_ID)
);
```

Discussion Questions

1. Why is an **ER Diagram essential** before creating a database?
2. How does **Schema Normalization** improve database performance?
3. What are the **challenges in designing relationships** for complex databases?

Conclusion

A **well-structured ER Diagram** and **optimized Schema Design** ensure **data integrity, efficiency, and scalability** in databases. By applying **normalization rules, indexing, and optimized SQL queries**, businesses can manage large-scale databases efficiently. 🚀

HANDLING DATA REDUNDANCY & DENORMALIZATION TECHNIQUES

CHAPTER 1: UNDERSTANDING DATA REDUNDANCY AND ITS CHALLENGES

Definition and Importance

Data redundancy refers to the **unnecessary duplication of data** in a database. While some redundancy is intentional for performance optimization, excessive redundancy can lead to **storage inefficiency, data inconsistency, and higher maintenance costs**.

Data redundancy typically occurs when:

- The same data is stored in multiple places.
- Relationships between tables are not well-defined.
- Database normalization rules (1NF, 2NF, 3NF, BCNF) are not properly implemented.

Problems Caused by Data Redundancy

1. **Increased Storage Costs:** Duplicate data requires more disk space.
2. **Data Inconsistency:** Updates in one location may not be reflected elsewhere.
3. **Higher Maintenance Effort:** Requires extra work to keep data synchronized.

4. **Performance Issues:** Redundant data can slow down queries due to unnecessary joins and increased index sizes.

Example of Data Redundancy

Consider an **E-Commerce database** where customer details are stored in both the **Orders** and **Customers** tables:

Orders Table (with Redundancy)

Order_ID	Customer_ID	Customer_Name	Customer_Email	Product	Amount
101	1001	Alice Smith	alice@email.com	Laptop	1200
102	1001	Alice Smith	alice@email.com	Mouse	50
103	1002	Bob Johnson	bob@email.com	Keyboard	100

Here, **Customer_Name** and **Customer_Email** are repeated for each order, causing redundancy.

Exercise

- Identify redundant columns in an existing database.
- Write an SQL query to **count duplicate records** in a table.

CHAPTER 2: TECHNIQUES TO HANDLE DATA REDUNDANCY

1. Applying Normalization to Minimize Redundancy

Database **normalization** is the best way to reduce redundancy while ensuring **data integrity and consistency**.

Solution for the E-Commerce Example

1. **Create a separate Customers table**:

```
CREATE TABLE Customers (
    Customer_ID INT PRIMARY KEY,
    Customer_Name VARCHAR(100),
    Customer_Email VARCHAR(100) UNIQUE
);
```

2. **Modify the Orders table to reference Customers**:

```
CREATE TABLE Orders (
    Order_ID INT PRIMARY KEY,
    Customer_ID INT,
    Product VARCHAR(100),
    Amount DECIMAL(10,2),
    FOREIGN KEY (Customer_ID) REFERENCES Customers(Customer_ID)
);
```

Now, customer details are stored in **one place**, reducing redundancy.

Exercise

- Normalize a **redundant dataset** by breaking it into **multiple tables**.
- Write a **JOIN query** to fetch data from normalized tables efficiently.

Chapter 3: Understanding Denormalization and When to Use It

Definition of Denormalization

Denormalization is the process of **intentionally introducing some redundancy** in a database to **improve query performance**. While normalization ensures **data integrity**, it sometimes slows down **read-heavy applications** due to multiple joins.

When Should You Use Denormalization?

1. **Read-heavy databases** where complex joins slow down queries.
2. **Data Warehousing** where fast reporting is required.
3. **Caching frequently accessed data** to reduce query execution time.
4. **Distributed databases** where reducing join operations improves performance.

Example of Denormalization in an E-Commerce Database

In a **high-traffic e-commerce platform**, querying order details requires multiple joins:

```
SELECT Orders.Order_ID, Customers.Customer_Name,
Customers.Customer_Email, Orders.Product, Orders.Amount

FROM Orders

JOIN Customers ON Orders.Customer_ID =
Customers.Customer_ID;
```

This **JOIN query can slow down** the system if the database has **millions of records**.

Denormalized Solution

Instead of fetching data from multiple tables, store customer details **inside the Orders table**:

Order_ID	Customer_ID	Customer_Name	Customer_Email	Product	Amount
101	1001	Alice Smith	alice@email.com	Laptop	1200
102	1001	Alice Smith	alice@email.com	Mouse	50
103	1002	Bob Johnson	bob@email.com	Keyboard	100

Now, we **avoid JOINs**, making queries **faster**, though at the cost of **data redundancy**.

Exercise

- Identify a table where **denormalization** could improve performance.

- Compare the **query execution time** before and after denormalization.

Chapter 4: Denormalization Techniques for Optimizing Query Performance

1. Precomputed Aggregates for Reporting

Instead of computing aggregates on the fly, store **precomputed values** in a separate table.

Example:

- **Normalized query:**

```
SELECT Customer_ID, SUM(Amount) AS Total_Spent FROM Orders GROUP BY Customer_ID;
```

- **Denormalized solution:** Maintain a **Customer_Spending** table:

```
CREATE TABLE Customer_Spending (
    Customer_ID INT PRIMARY KEY,
    Total_Spent DECIMAL(10,2)
);
```

Now, queries fetch precomputed results **instantly**.

2. Using Materialized Views for Faster Access

A **Materialized View** stores query results as a physical table, reducing computation time.

```
CREATE MATERIALIZED VIEW OrderSummary AS

SELECT Customer_ID, COUNT(Order_ID) AS Total_Orders,
SUM(Amount) AS Total_Spent

FROM Orders

GROUP BY Customer_ID;
```

Queries on **OrderSummary** will be **much faster** than recalculating totals dynamically.

3. Adding Redundant Columns for Query Optimization

To **avoid frequent JOIN operations**, store frequently accessed columns together.

Example:

Instead of joining **Orders** and **Customers** tables every time, store **Customer_Name** directly inside the **Orders** table.

```
ALTER TABLE Orders ADD COLUMN Customer_Name
VARCHAR(100);
```

Now, queries can be optimized to:

```
SELECT Order_ID, Customer_Name, Product, Amount FROM
Orders;
```

This eliminates **JOINs**, improving performance.

Exercise

- Create a **Materialized View** for a slow-running query.

- Compare query execution speed **before and after** using precomputed aggregates.

CHAPTER 5: CASE STUDY – OPTIMIZING A HIGH-TRAFFIC E-COMMERCE DATABASE

Scenario

An e-commerce company with **millions of orders** faces **slow query performance** when customers check their **order history**.

Step 1: Identifying Issues

- Queries involve multiple **JOINs** (Orders + Customers + Products).
- Aggregations for **total spending and order count** slow down reports.

Step 2: Applying Denormalization Techniques

1. **Precomputed Aggregates:**
    - Store **total spending per customer** in a **Customer_Spending** table.
2. **Materialized Views for Faster Reporting:**

CREATE MATERIALIZED VIEW FastOrderSummary AS

SELECT Customer_ID, COUNT(Order_ID) AS Total_Orders, SUM(Amount) AS Total_Spent

FROM Orders

GROUP BY Customer_ID;

3. **Adding Redundant Columns to Reduce JOINs:**
    - Store **Customer_Name** inside the **Orders** table.

Step 3: Results After Optimization

✅ Query execution **time reduced by 70%**.
✅ Reporting dashboards load **5x faster**.
✅ Users experience **better performance** when checking order history.

Discussion Questions

1. When should **normalization** be prioritized over **denormalization**?
2. How do **Materialized Views** improve query performance?
3. What are the **trade-offs of storing redundant data**?

Conclusion

Understanding **when to normalize vs. when to denormalize** is critical for **database optimization**. While normalization ensures **data integrity**, denormalization enhances **query speed and scalability**

MySQL User Roles & Permissions

Chapter 1: Introduction to MySQL User Roles and Permissions

Definition and Importance

In MySQL, **user roles and permissions** are essential for **database security and access control**. They define **who can access the database, what actions they can perform, and which data they can modify**. Proper user management ensures **data security, prevents unauthorized changes, and protects against potential cyber threats**.

Permissions in MySQL control actions such as:

- **Reading Data** (SELECT queries).
- **Modifying Data** (INSERT, UPDATE, DELETE).
- **Creating or Altering Tables** (CREATE, ALTER, DROP).
- **Granting Permissions to Other Users** (GRANT).

Understanding and managing MySQL **user roles and permissions** is crucial for **database administrators (DBAs)**, ensuring data integrity while allowing users the necessary access to perform their tasks efficiently.

Chapter 2: Understanding MySQL User Roles

1. What are MySQL User Roles?

A **role** in MySQL is a collection of **privileges** assigned to a user or group of users. Instead of granting permissions **individually to each user**, roles simplify permission management by grouping related permissions together.

2. Types of MySQL Users

Different types of users require varying levels of access:

- **Database Administrator (DBA)** – Full control over the database, including creating users and assigning privileges.
- **Developer** – Access to tables, procedures, and views but limited administrative control.
- **Analyst** – Read-only access for reporting and data analysis.
- **Application User** – Limited access for executing specific queries in an application.

3. Example of Creating a User Role

```
CREATE ROLE developer_role;

GRANT SELECT, INSERT, UPDATE, DELETE ON mydatabase.* TO developer_role;

GRANT EXECUTE ON PROCEDURE mydatabase.* TO developer_role;
```

This **developer_role** can now be assigned to users instead of granting individual privileges.

Exercise

- Identify different types of users in your organization.
- Create **at least two user roles** with appropriate permissions.

CHAPTER 3: MYSQL PRIVILEGES AND PERMISSION TYPES

1. Global vs. Database-Specific Permissions

Permissions in MySQL can be assigned at different levels:

- **Global Level:** Grants access to all databases.
- **Database Level:** Grants access to a specific database.
- **Table Level:** Grants access to a specific table within a database.
- **Column Level:** Grants access to specific columns in a table.
- **Stored Procedure Level:** Grants permission to execute stored procedures.

2. Types of MySQL Permissions

Permission	Description
ALL PRIVILEGES	Grants all privileges (DANGEROUS—use with caution).
SELECT	Allows reading data from tables.
INSERT	Allows adding new records to tables.
UPDATE	Allows modifying existing records.

DELETE	Allows removing records from tables.
CREATE	Allows creating new tables or databases.
DROP	Allows deleting tables or databases.
ALTER	Allows modifying table structure.
GRANT OPTION	Allows granting permissions to other users.

3. Granting and Revoking Permissions

Granting Permissions to a User

GRANT SELECT, INSERT ON mydatabase.* TO 'developer'@'localhost';

Revoking Permissions from a User

REVOKE INSERT ON mydatabase.* FROM 'developer'@'localhost';

These commands **control access** to ensure users only have the permissions they need.

Exercise

- Create a **new user** and grant them **SELECT and UPDATE permissions**.
- Revoke **UPDATE permission** and verify the changes.

CHAPTER 4: MANAGING MYSQL USERS AND ROLES

1. Creating and Deleting Users

Creating a New User

```
CREATE USER 'john_doe'@'localhost' IDENTIFIED BY 'SecurePassword123';
```

Deleting a User

```
DROP USER 'john_doe'@'localhost';
```

These commands help in **adding and removing users** from the database system.

2. Assigning and Removing Roles

Assigning a Role to a User

```
GRANT developer_role TO 'john_doe'@'localhost';
```

Removing a Role from a User

```
REVOKE developer_role FROM 'john_doe'@'localhost';
```

Using roles ensures **easier user management** without assigning individual permissions.

3. Checking User Privileges

To view a user's assigned privileges:

```
SHOW GRANTS FOR 'john_doe'@'localhost';
```

This command **displays all privileges** assigned to the user.

Exercise

- Create a **new user** with a strong password.

- Assign them **a specific role** and check their granted privileges.

Chapter 5: Case Study – Securing an E-Commerce Database

Scenario

An e-commerce company has the following roles:

1. **DBA:** Full access to manage the database.
2. **Developers:** Can modify product data but cannot delete records.
3. **Sales Analysts:** Can read sales reports but cannot modify data.
4. **Application Users:** Can only insert new orders.

Step 1: Creating User Roles

```
CREATE ROLE db_admin;

CREATE ROLE developer;

CREATE ROLE analyst;

CREATE ROLE app_user;
```

Step 2: Assigning Permissions

```
GRANT ALL PRIVILEGES ON ecom_db.* TO db_admin;
```

GRANT SELECT, INSERT, UPDATE ON ecom_db.products TO developer;

GRANT SELECT ON ecom_db.sales TO analyst;

GRANT INSERT ON ecom_db.orders TO app_user;

Step 3: Creating Users and Assigning Roles

CREATE USER 'alice'@'localhost' IDENTIFIED BY 'AliceSecure123';

GRANT developer TO 'alice'@'localhost';

CREATE USER 'bob'@'localhost' IDENTIFIED BY 'BobSecure123';

GRANT analyst TO 'bob'@'localhost';

CREATE USER 'charlie'@'localhost' IDENTIFIED BY 'CharlieSecure123';

GRANT app_user TO 'charlie'@'localhost';

Step 4: Verifying Security

SHOW GRANTS FOR 'alice'@'localhost';

SHOW GRANTS FOR 'bob'@'localhost';

SHOW GRANTS FOR 'charlie'@'localhost';

Outcome

✅ **DBA manages the entire database.**
✅ **Developers can modify product data but cannot delete.**
✅ **Sales analysts can only read sales data.**
✅ **Application users can only add new orders.**

Discussion Questions

1. Why is it important to define **roles instead of assigning permissions directly** to users?
2. How does **revoking unnecessary permissions** enhance security?
3. What are the risks of granting **ALL PRIVILEGES** to a user?

Conclusion

Managing **MySQL user roles and permissions** is essential for **database security and efficiency**. By using **roles, granting only necessary privileges, and monitoring user activity**, organizations can protect **sensitive data while maintaining operational flexibility**.

🚀 **Next Steps:**

- Implement **user roles in your own database**.
- Use SHOW GRANTS to audit user permissions.
- Apply **role-based access control (RBAC)** for enhanced security.

PREVENTING SQL INJECTION ATTACKS

CHAPTER 1: INTRODUCTION TO SQL INJECTION ATTACKS

Definition and Importance

SQL Injection (SQLi) is one of the most **dangerous security vulnerabilities** in web applications. It occurs when **malicious SQL code is injected into user input fields** to manipulate or gain unauthorized access to a database. Attackers can **steal sensitive data, modify records, or even delete entire databases** if proper security measures are not in place.

SQL Injection is listed as one of the **OWASP Top 10 security risks**, making it a **critical concern for database administrators, developers, and security professionals**. Proper prevention techniques ensure the protection of **confidential user data, financial records, and business information**.

How SQL Injection Works

Attackers typically exploit **user input fields**, URLs, or cookies to insert **malicious SQL queries**. If the application does not properly validate input, the attacker can execute **unauthorized SQL commands**.

Example of a Basic SQL Injection Attack

A vulnerable login query:

```
SELECT * FROM users WHERE username = 'admin' AND password = 'password';
```

If an attacker enters the following input in the **username field**:

' OR '1'='1

The query becomes:

SELECT * FROM users WHERE username = '' OR '1'='1' AND password = 'password';

Since '1'='1' is always **true**, the attacker **bypasses authentication**, gaining access to the system.

Exercise

- Write a vulnerable SQL query and modify it to prevent SQL Injection.
- Identify **possible injection points** in an application.

CHAPTER 2: TYPES OF SQL INJECTION ATTACKS

1. Classic SQL Injection

This occurs when user input is **directly inserted** into a SQL query without validation.

Example

SELECT * FROM users WHERE username = 'admin' AND password = 'password';

If an attacker inputs:

' OR '1'='1' --

The query becomes:

SELECT * FROM users WHERE username = '' OR '1'='1' -- ' AND password = 'password';

Here, -- comments out the rest of the SQL statement, bypassing authentication.

2. Blind SQL Injection

In **Blind SQL Injection**, attackers do not see database error messages but can infer data by observing **application behavior**.

Example

An attacker tries:

' OR IF((SELECT database())='mydb', SLEEP(5), NULL) --

If the page takes **5 seconds** to respond, they confirm that the database name is **mydb**.

3. Time-Based SQL Injection

Attackers use time delays to determine whether an SQL query is vulnerable.

Example

' OR IF(1=1, SLEEP(10), NULL) --

If the page **delays by 10 seconds**, the application is vulnerable.

Exercise

- Simulate **Blind SQL Injection** using different payloads.
- Test **time-based SQL Injection** on a test database.

CHAPTER 3: BEST PRACTICES TO PREVENT SQL INJECTION

1. Using Prepared Statements (Parameterized Queries)

Prepared statements separate **SQL logic from user input**, preventing attackers from modifying queries.

Example of a Secure Query (PHP & MySQL)

```
$stmt = $conn->prepare("SELECT * FROM users WHERE username = ? AND password = ?");
$stmt->bind_param("ss", $username, $password);
$stmt->execute();
```

Here, ? acts as a **placeholder**, ensuring user input is treated as **data, not code**.

2. Input Validation and Sanitization

- **Reject unexpected input** (e.g., special characters, SQL keywords).
- **Use regular expressions** to allow only valid input formats.
- **Convert input to a safe format** before processing.

Example of Input Validation (Python)

```
import re
```

```
def validate_input(user_input):
    if not re.match("^[a-zA-Z0-9_]+$", user_input):
        return False
    return True
```

3. Limiting Database Privileges

Even if an attacker manages to inject SQL, **restricting user privileges** prevents them from causing significant damage.

Best Practices for Database Privileges

- **Use the principle of least privilege (PoLP)** – restrict user access to only necessary operations.
- **Avoid using root/admin database accounts** for application queries.
- **Grant specific permissions** instead of ALL PRIVILEGES.

Example of Restricted User Creation

```
CREATE USER 'app_user'@'localhost' IDENTIFIED BY 'StrongPassword!';

GRANT SELECT, INSERT ON mydatabase.* TO 'app_user'@'localhost';
```

Here, the user **app_user** cannot modify or delete records, reducing potential damage.

Exercise

- Modify a SQL query to use **prepared statements** instead of raw queries.
- Create a **low-privileged database user** and test what queries they can execute.

Chapter 4: Additional Security Measures

1. Using Web Application Firewalls (WAFs)

A **Web Application Firewall (WAF)** detects and blocks **malicious SQL injection attempts** in real-time.

Popular WAFs for SQL Injection Protection

- **ModSecurity** – An open-source WAF for Apache, Nginx, and IIS.
- **Cloudflare WAF** – Provides **built-in SQL Injection protection**.

2. Encrypting Sensitive Data

Even if an attacker **gains access** to the database, encrypted data remains **unreadable**.

Example of Hashing Passwords in MySQL

```
INSERT INTO users (username, password) VALUES ('admin', SHA2('securepassword', 256));
```

Instead of storing **plain text passwords**, use **SHA-256 encryption**.

3. Regular Security Audits

- Perform **penetration testing** on databases.
- **Log all SQL queries** to detect suspicious activity.
- Monitor **database access logs** for unauthorized attempts.

Exercise

- Implement **password hashing** in a login system.
- Set up **database logging** to track SQL queries.

CHAPTER 5: CASE STUDY – PREVENTING SQL INJECTION IN AN E-COMMERCE WEBSITE

Scenario

An **e-commerce platform** suffered an SQL Injection attack, allowing an attacker to:

- Extract **customer credit card information**.
- Modify **order details** without authentication.
- Delete **critical business data**.

Step 1: Identifying Vulnerabilities

- The application used **concatenated SQL queries**:

  $query = "SELECT * FROM users WHERE email = '" . $_POST['email'] . "'";

- User input **was not validated or sanitized**.

- The **database user had excessive privileges**, allowing DELETE operations.

Step 2: Implementing Security Fixes

✅ **Use Prepared Statements**

```
$stmt = $conn->prepare("SELECT * FROM users WHERE email = ?");
$stmt->bind_param("s", $_POST['email']);
$stmt->execute();
```

✅ **Sanitize Inputs with Regular Expressions**

```
$email = filter_var($_POST['email'], FILTER_VALIDATE_EMAIL);
```

✅ **Restrict Database User Privileges**

```
REVOKE DELETE ON orders FROM 'app_user'@'localhost';
```

✅ **Enable Web Application Firewall (WAF)**
Deployed **ModSecurity WAF**, blocking **malicious SQL Injection attempts**.

Step 3: Results After Fixes

- 🚀 **No further SQL Injection attacks detected.**
- 🔒 **Customer data remained secure.**
- ⚡ **Performance improved by reducing unnecessary database access.**

Discussion Questions

1. Why are **prepared statements** better than string concatenation?
2. How does **input validation** prevent SQL Injection?
3. What are the risks of **granting all privileges** to a database user?

Conclusion

Preventing **SQL Injection** requires a **multi-layered security approach**, including **prepared statements, input validation, user privilege management, encryption, and firewalls**. Organizations must **continuously monitor and test** their applications to stay protected against evolving threats.

🚀 **Next Steps:**

- Implement **prepared statements** in your applications.
- Test for **SQL Injection vulnerabilities** using tools like **SQLmap**.
- Conduct **regular security audits** to prevent future attacks.

ENCRYPTING DATA IN MYSQL

CHAPTER 1: INTRODUCTION TO DATA ENCRYPTION IN MYSQL

Definition and Importance

Data encryption is a **crucial security measure** that ensures **sensitive information** stored in a database remains **protected from unauthorized access**. Encryption transforms **plain text** into **unreadable ciphertext**, which can only be decrypted using a specific key.

In MySQL, encryption is essential for:

- **Protecting sensitive data** such as passwords, credit card numbers, and personal details.
- **Preventing data breaches** in case of unauthorized access.
- **Complying with security standards** like **GDPR, HIPAA, and PCI DSS**.
- **Ensuring data integrity** by preventing unauthorized modifications.

MySQL provides **multiple encryption techniques**, including **hashing, symmetric encryption, and asymmetric encryption**, to secure data both **at rest** (stored data) and **in transit** (data being transmitted).

Exercise

- Identify **sensitive data** in your database that should be encrypted.

- Research security **regulations** that require encryption.

CHAPTER 2: TYPES OF DATA ENCRYPTION IN MYSQL

1. Hashing – One-Way Encryption

Hashing is a **one-way encryption technique** that converts data into a fixed-length **hash value**. It is commonly used for **storing passwords** securely.

Example of Hashing in MySQL

```
SELECT SHA2('SecurePassword123', 256);
```

- **SHA2()** generates a **hashed password** that cannot be reversed.

Plain Password	SHA2 (256-bit Hash)
Secure123	3a7bd3e2360...

2. Symmetric Encryption – Two-Way Encryption

Symmetric encryption **uses the same key** to encrypt and decrypt data. MySQL provides the **AES_ENCRYPT()** and **AES_DECRYPT()** functions for this purpose.

Example of Symmetric Encryption in MySQL

```
SELECT AES_ENCRYPT('SensitiveData', 'encryption_key');
```

```
SELECT AES_DECRYPT(AES_ENCRYPT('SensitiveData',
'encryption_key'), 'encryption_key');
```

- **AES_ENCRYPT()** converts plain text into ciphertext.
- **AES_DECRYPT()** converts ciphertext back into readable data using the same key.

3. Asymmetric Encryption – Public & Private Keys

Asymmetric encryption uses **two keys (public and private)** for encryption and decryption. It is used for **secure data transmission** but is computationally expensive.

- MySQL does not natively support asymmetric encryption but **OpenSSL** or **third-party libraries** can be used.

Exercise

- Implement **SHA-256 hashing** for user passwords in MySQL.
- Encrypt and decrypt sample data using **AES_ENCRYPT()**.

CHAPTER 3: IMPLEMENTING DATA ENCRYPTION IN MYSQL

1. Encrypting Sensitive Data Using AES Encryption

To **secure personal details** (e.g., Social Security Number), store **encrypted values** in the database.

Example – Encrypting and Storing Data

```
INSERT INTO users (user_id, name, ssn_encrypted)

VALUES (1, 'John Doe', AES_ENCRYPT('123-45-6789',
'my_secret_key'));
```

Example – Decrypting Data

SELECT name, AES_DECRYPT(ssn_encrypted, 'my_secret_key') AS ssn FROM users;

✅ **Benefit**: Even if the database is **compromised**, encrypted values remain **unreadable**.

2. Encrypting Data Before Insertion (Application-Level Encryption)

Instead of encrypting data **inside MySQL**, use **application-layer encryption** (e.g., PHP, Python).

Example – Encrypting Data in PHP Before Storing

$encryptedData = openssl_encrypt("SensitiveInfo", "AES-256-CBC", "my_secret_key");

- **This method prevents SQL Injection vulnerabilities.**

Exercise

- Encrypt and store **email addresses** using AES encryption.
- Retrieve encrypted email addresses and decrypt them in MySQL.

CHAPTER 4: SECURING STORED PASSWORDS WITH HASHING

1. Why Hashing is Better Than Encryption for Passwords

- **Encryption** is **reversible** (data can be decrypted if the key is exposed).

- **Hashing** is **one-way** (cannot be reversed), making it **ideal for storing passwords**.

2. Hashing Passwords with MySQL SHA2()

```
INSERT INTO users (user_id, username, password_hash)
VALUES (1, 'john_doe', SHA2('SuperSecurePass', 256));
```

3. Verifying a Hashed Password

To authenticate a user:

```
SELECT * FROM users WHERE username = 'john_doe' AND password_hash = SHA2('SuperSecurePass', 256);
```

✅ **Benefit**: Even if a hacker steals the **hashed password**, they **cannot retrieve the original password**.

Exercise

- Create a **user authentication system** using hashed passwords.
- Try to **reverse** a hashed password (Hint: It's impossible!).

Chapter 5: Case Study – Securing an E-Commerce Database with Encryption

Scenario

An **e-commerce company** faced a **data breach** where **customer credit card details** were exposed. The company decided to **implement encryption** to enhance security.

Step 1: Identifying Sensitive Data

The following columns need encryption:

1. **Credit Card Numbers**
2. **Customer Social Security Numbers (SSN)**
3. **Passwords**

Step 2: Implementing Encryption & Hashing

✅ Encrypting Credit Card Numbers

```
UPDATE customers
SET credit_card_encrypted =
AES_ENCRYPT(credit_card_number, 'encryption_key');
```

✅ Hashing Passwords

```
UPDATE users
SET password_hash = SHA2(password_plain, 256);
```

✅ Using Least Privilege for Database Users

```
REVOKE SELECT ON customers FROM
'app_user'@'localhost';
```

Step 3: Results After Implementation

- 🚀 **Credit card details are now unreadable** even if stolen.
- 🔒 **Passwords are stored securely**, reducing hacking risks.
- ⚡ **Database performance remains optimized** with indexed searches.

Discussion Questions

1. Why should **passwords be hashed** instead of encrypted?
2. What are the **advantages of using AES encryption** for credit card data?
3. How can encryption impact **database performance**?

Conclusion

Encrypting **data in MySQL** is a **critical security measure** to **protect sensitive information** from cyber threats. Using **hashing for passwords**, **AES encryption for personal data**, and **privileged user access** ensures **data confidentiality and compliance with regulations**.

🚀 **Next Steps:**

- Implement **AES encryption** for customer **emails and phone numbers**.
- Perform **penetration testing** to assess encryption effectiveness.
- Explore **column-level encryption** with **MySQL Enterprise Edition**.

ASSIGNMENT 3:

✅ DESIGN A SECURE DATABASE SCHEMA FOR A HOSPITAL MANAGEMENT SYSTEM, ENSURING PROPER NORMALIZATION AND SECURITY BEST PRACTICES.

Step-by-Step Guide: Designing a Secure Database Schema for a Hospital Management System (HMS)

A **Hospital Management System (HMS)** requires a **secure, well-structured** database to efficiently store and manage **patient records, doctor information, appointments, prescriptions, billing, and staff management** while ensuring **data integrity, security, and compliance** with regulations such as **HIPAA and GDPR**.

Step 1: Identifying Core Entities and Relationships

Entities in HMS Database

1. **Patients** – Stores patient details.
2. **Doctors** – Stores doctor information.
3. **Appointments** – Manages patient-doctor appointments.
4. **Medical Records** – Stores patient diagnoses, treatments, and reports.
5. **Prescriptions** – Stores prescribed medications.
6. **Billing** – Manages hospital billing and payments.
7. **Staff** – Stores details of hospital employees (non-doctors).
8. **Users** – Stores login credentials with access roles (Admins, Doctors, Staff).

Step 2: Designing Normalized Schema (Applying 1NF, 2NF, 3NF, BCNF)

1. Patients Table (1NF, 2NF, 3NF Compliant)

Stores unique patient details with a **primary key (Patient_ID)**.

```
CREATE TABLE Patients (
    Patient_ID INT PRIMARY KEY AUTO_INCREMENT,
    First_Name VARCHAR(100) NOT NULL,
    Last_Name VARCHAR(100) NOT NULL,
    DOB DATE NOT NULL,
    Gender ENUM('Male', 'Female', 'Other') NOT NULL,
    Contact_Number VARCHAR(15) NOT NULL UNIQUE,
    Email VARCHAR(255) UNIQUE,
    Address TEXT,
    Emergency_Contact VARCHAR(15),
    Created_At TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

✅ Security Considerations:

- **Patient_ID** is auto-incremented for uniqueness.

- **Contact numbers and emails** are unique to prevent duplication.
- **Timestamping records** for tracking.

---

2. Doctors Table

Stores doctor details, each uniquely identified by **Doctor_ID**.

```
CREATE TABLE Doctors (
    Doctor_ID INT PRIMARY KEY AUTO_INCREMENT,
    First_Name VARCHAR(100) NOT NULL,
    Last_Name VARCHAR(100) NOT NULL,
    Specialization VARCHAR(255) NOT NULL,
    Contact_Number VARCHAR(15) NOT NULL UNIQUE,
    Email VARCHAR(255) UNIQUE,
    Qualification VARCHAR(255) NOT NULL,
    Years_Experience INT NOT NULL,
    Created_At TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

✅ **Normalization Compliance:**

- No **repeating fields** (1NF).

- Every column depends on **Doctor_ID** (2NF).
- No **transitive dependencies** (3NF).

✅ **Security Measures:**

- **Emails and contact numbers are unique**.
- **Restrict unauthorized access** to doctor details.

---

3. Appointments Table

Manages appointments between **patients and doctors**.

```
CREATE TABLE Appointments (
    Appointment_ID INT PRIMARY KEY AUTO_INCREMENT,
    Patient_ID INT NOT NULL,
    Doctor_ID INT NOT NULL,
    Appointment_Date DATETIME NOT NULL,
    Status ENUM('Scheduled', 'Completed', 'Cancelled') DEFAULT 'Scheduled',
    Created_At TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (Patient_ID) REFERENCES Patients(Patient_ID) ON DELETE CASCADE,
    FOREIGN KEY (Doctor_ID) REFERENCES Doctors(Doctor_ID) ON DELETE CASCADE
```

);

✅ **Security Enhancements:**

- **Foreign keys ensure relational integrity.**
- **ON DELETE CASCADE** ensures patient/doctor deletion removes related records.
- **Appointment status tracking** prevents duplication.

---

4. Medical Records Table

Stores patient diagnosis and treatment details.

```
CREATE TABLE Medical_Records (
    Record_ID INT PRIMARY KEY AUTO_INCREMENT,
    Patient_ID INT NOT NULL,
    Doctor_ID INT NOT NULL,
    Diagnosis TEXT NOT NULL,
    Treatment TEXT NOT NULL,
    Prescribed_Date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (Patient_ID) REFERENCES Patients(Patient_ID) ON DELETE CASCADE,
    FOREIGN KEY (Doctor_ID) REFERENCES Doctors(Doctor_ID) ON DELETE CASCADE
```

);

✅ **Security Features:**

- **Confidentiality enforced by restricting access to authorized users.**
- **Access logs should be maintained for record modifications.**

---

5. Prescriptions Table

Stores prescribed medications for patients.

```
CREATE TABLE Prescriptions (
    Prescription_ID INT PRIMARY KEY AUTO_INCREMENT,
    Record_ID INT NOT NULL,
    Medicine_Name VARCHAR(255) NOT NULL,
    Dosage VARCHAR(100) NOT NULL,
    Duration VARCHAR(50),
    FOREIGN KEY (Record_ID) REFERENCES
Medical_Records(Record_ID) ON DELETE CASCADE
);
```

✅ **Security Practices:**

- **Link prescriptions with medical records** to ensure authenticity.

- **Restrict modifications** to prescriptions post-issuance.

---

6. Billing Table

Stores billing and payment transactions.

```
CREATE TABLE Billing (
    Bill_ID INT PRIMARY KEY AUTO_INCREMENT,
    Patient_ID INT NOT NULL,
    Amount DECIMAL(10,2) NOT NULL,
    Payment_Status ENUM('Pending', 'Paid', 'Cancelled') DEFAULT 'Pending',
    Payment_Date TIMESTAMP NULL,
    FOREIGN KEY (Patient_ID) REFERENCES Patients(Patient_ID) ON DELETE CASCADE
);
```

✅ **Security Measures:**

- **Encryption of sensitive billing data.**
- **Payment audits for fraud detection.**

---

7. Users & Role-Based Access Control (RBAC)

Manages **login authentication and role-based access**.

```
CREATE TABLE Users (
```

```
    User_ID INT PRIMARY KEY AUTO_INCREMENT,
    Username VARCHAR(100) UNIQUE NOT NULL,
    Password_Hash VARCHAR(256) NOT NULL,
    Role ENUM('Admin', 'Doctor', 'Staff', 'Patient') NOT NULL,
    Created_At TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

✅ **Security Features:**

- **Password hashing using SHA-256.**
- **Role-based access control (RBAC) prevents unauthorized access.**

Step 3: Implementing Security Best Practices

1. Encrypting Sensitive Data

Use **AES encryption** for storing critical information.

```
UPDATE Patients
SET Contact_Number = AES_ENCRYPT('9876543210', 'encryption_key')
WHERE Patient_ID = 1;
```

✅ **Ensures only authorized users can decrypt the data.**

---

2. Hashing Passwords for Secure Authentication

```
INSERT INTO Users (Username, Password_Hash, Role)
VALUES ('admin', SHA2('StrongPassword123', 256), 'Admin');
```

✅ **Prevents password exposure in case of data leaks.**

---

3. Granting Minimum Privileges (Principle of Least Privilege)

Restrict database access to **only required users**.

```
CREATE USER 'hms_app'@'localhost' IDENTIFIED BY 'SecurePassword!';
GRANT SELECT, INSERT, UPDATE ON hospital_db.* TO 'hms_app'@'localhost';
```

✅ **Prevents unauthorized users from modifying records.**

---

4. Regular Security Audits

Enable **logging** to track user activity.

```
SET GLOBAL general_log = 'ON';
SHOW VARIABLES LIKE 'general_log%';
```

✅ **Detects suspicious queries or unauthorized data access.**

Step 4: Case Study – Secure HMS Implementation for a Multi-Specialty Hospital

Scenario

A multi-specialty hospital experienced:

- Unauthorized access to **patient records**.
- Doctors accessing **billing details** unnecessarily.
- Patients modifying **appointment records**.

Implemented Solutions

✅ **Role-based access control (RBAC)** prevents unauthorized access.
✅ **AES encryption applied** to patient records.
✅ **Restricted database privileges** to specific roles.
✅ **Regular security audits & logging** to detect unauthorized access.

Outcome

🚀 **Improved data security & compliance** with regulations.
🔒 **Patients' records remain protected from unauthorized modifications.**
⚡ **Faster system performance with optimized queries.**

Conclusion

A **secure Hospital Management System** database requires:

- ✔ **Proper normalization to prevent redundancy.**
- ✔ **Encryption for securing patient-sensitive data.**
- ✔ **Role-based access control (RBAC) to restrict access.**
- ✔ **Regular audits to detect security vulnerabilities.**

🚀 **Next Steps:**

- Implement **API security measures** for external integrations.
- Use **two-factor authentication (2FA) for admin access.**
- Perform **penetration testing** to identify security loopholes.

Let me know if you need **hands-on SQL scripts or real-world database optimization!** 🔒💾