UNIT 4 – MySQL Practical Implementation

Unit Overview

This unit focuses on **practical, real-world implementation** of database concepts using MySQL. You'll install MySQL, create databases, design tables, establish relationships, write stored procedures, and build automation using triggers — ending with a **Mini Project: Library Management System**.

Unit Topics

Subtopic Code	Topic Name	Subtopics
4.1	**MySQL Installation & Setup**	Installing MySQL & Workbench, Connecting, Navigating Databases
4.2	**MySQL Data Types**	Numeric, String, Date/Time, Boolean, Choosing Appropriate Data Types
4.3	**Creating Databases & Tables**	CREATE DATABASE, USE, CREATE TABLE, Applying Constraints
4.4	**Foreign Keys & Relationships**	Referential Integrity, ON DELETE/ON UPDATE CASCADE
4.5	**Stored Procedures & Functions**	Syntax, Parameters, Execution, Real-life Examples
4.6	**Triggers**	Concept, BEFORE vs AFTER Triggers, Syntax & Use Cases
4.7	**Backup & Restore**	Using MySQL Workbench & mysqldump Tool
4.8	**Mini Project**	*Library Management System* integrating relationships, stored procs, and triggers

Learning Outcome (After Unit 4)

By the end of this unit, learners will be able to:

- Set up and use MySQL efficiently on their systems.
- Create and manage databases, tables, and relationships.
- Use stored procedures, triggers, and functions for automation.
- Maintain backups and restore databases confidently.
- Implement a fully working **mini project** using all learned concepts.

4.1 MySQL Installation & Setup

Subtopics

- Installing MySQL & MySQL Workbench
- Connecting to MySQL Server
- Navigating Databases in MySQL Workbench

Learning Outcome

After this topic, you will be able to:

- Install MySQL and MySQL Workbench on your system.
- Connect to the MySQL server and manage databases.
- Navigate schemas, tables, and queries inside Workbench.

1. Installing MySQL

MySQL Components

Component	Purpose
MySQL Server	The actual database engine that stores and manages data.
MySQL Workbench	GUI tool for writing queries, designing databases, and visualizing data.
MySQL Shell / Command-Line Client	CLI interface for running SQL statements.
Connector Drivers	Libraries that connect applications (Java, Python, etc.) to MySQL.

Steps for Installation (Windows/macOS/Linux)

Step 1:

Visit → https://dev.mysql.com/downloads/

Step 2:

Download **MySQL Installer** (Community Edition).

Step 3:

During setup:

- Select **Server + Workbench** installation.
- Choose **Developer Default** configuration.
- Set a **root password** (remember it for connection).

Step 4:

Complete installation → launch **MySQL Workbench**.

2. Connecting to MySQL Server

1. Open **MySQL Workbench**.
2. Click the default connection (usually named *Local Instance MySQL*).
3. Enter your **root password** → Click **OK**.
4. The Workbench interface opens with:
    - **Navigator Pane:** Schemas, tables, views, routines, etc.
    - **SQL Editor:** For executing SQL queries.
    - **Output Pane:** For query results and messages.

Connection Verification

In the SQL editor, type:

SELECT VERSION();

✅ Displays MySQL version — confirming successful setup.

3. Navigating Databases in Workbench

Task	Command	Purpose
View all databases	SHOW DATABASES;	Lists all available databases.
Create new database	CREATE DATABASE college;	Creates new schema.
Use a database	USE college;	Switch context to that database.
View all tables	SHOW TABLES;	Lists tables in current schema.
Describe a table	DESCRIBE student;	Displays column details.
Exit	QUIT;	Ends the session.

Example Setup Session

```
-- Create Database
CREATE DATABASE college;

-- Use Database
USE college;

-- Create Table
```

```
CREATE TABLE student (
    roll_no INT PRIMARY KEY,
    name VARCHAR(50),
    department VARCHAR(30)
);

-- Insert Data
INSERT INTO student VALUES (101, 'Kavya', 'CSE');

-- View Data
SELECT * FROM student;
```

Output:

roll_no	name	department
101	Kavya	CSE

4. Common MySQL Workbench Areas

Section	Purpose
Navigator → SCHEMAS	Lists databases and tables.
SQL Editor	Used to write and execute SQL queries.
Result Grid	Displays query results.
Action Buttons	Run, Stop, Export results.
Status Bar	Connection and query execution info.

5. Troubleshooting Common Issues

Can’t connect to MySQL	Server not running	Open **Services → MySQL → Start Service**
Forgot root password	Misconfigured	Re-run installer or reset password via shell
Port conflict	Default port 3306 in use	Change port in my.ini configuration
Query errors	Syntax or database not selected	Use USE database_name; before running queries

6. Verifying MySQL Setup (Checklist)

✅ MySQL Server installed and running

✅ MySQL Workbench opens successfully

✅ You can connect with root credentials

✅ Able to create and view databases

✅ Can execute basic SQL queries

Hands-On Practice

1. Launch **MySQL Workbench** and connect to your server.
2. Create a new schema named practice_db.
3. Inside it, create a table students(roll_no, name, dept).
4. Insert at least 3 records.
5. Query all records using SELECT * FROM students;.

Checkpoint Review

1. What are the components of MySQL installation?
2. How do you verify a successful connection in MySQL Workbench?
3. Which command is used to list all databases?
4. What could cause a “Cannot connect to MySQL” error?
5. What’s the default MySQL port number?

Reflection Prompt

If you were setting up MySQL in a training lab for multiple students, what configurations or settings would you standardize to ensure smooth practice sessions?

Summary

- MySQL setup includes **Server**, **Workbench**, and **Client Tools**.
- **Workbench** offers a GUI for writing queries and managing databases.
- The connection requires root authentication on **port 3306** by default.
- Always verify setup by running SELECT VERSION();.
- Common commands: CREATE DATABASE, USE, SHOW TABLES, DESCRIBE, SELECT.

4.2 MySQL Data Types

Subtopics

- Numeric Data Types
- String Data Types
- Date and Time Data Types
- Boolean Type
- Choosing the Right Data Type

Learning Outcome

After completing this topic, you will be able to:

- Identify and use the correct MySQL data type for each kind of information.
- Understand how data type selection affects performance and memory usage.
- Create optimized table schemas based on real-life data requirements.

1. Concept of Data Types

A **data type** defines the **kind of value** a column can hold — such as numbers, text, dates, or true/false.

Choosing the **correct data type** ensures:

- Efficient memory use
- Accurate data storage
- Better query performance
- Preventing invalid data

2. Numeric Data Types

Used for numbers — both whole and decimal values.

Type	Bytes	Range (Signed)	Use Case
TINYINT	1	-128 to 127	Status flags, Boolean values
SMALLINT	2	-32,768 to 32,767	Age, small counters
MEDIUMINT	3	~ -8 million to 8 million	Moderate counters
INT or INTEGER	4	~ -2 billion to 2 billion	IDs, population, quantities
BIGINT	8	Very large numbers	Bank balances, transaction IDs
FLOAT	4	Approx. precision	Scientific data, low-precision decimals
DOUBLE	8	High-precision decimals	Scientific calculations
DECIMAL(m,n)	Variable	Exact precision (m digits total, n after decimal)	Currency, billing, finance

Example:

```
CREATE TABLE products (
    product_id INT PRIMARY KEY,
    price DECIMAL(10,2),
    quantity SMALLINT
);
```

✅ DECIMAL(10,2) stores prices like 12345.67 accurately.

3. String Data Types

Used for textual data such as names, addresses, or descriptions.

Type	Maximum Size	Use Case
CHAR(n)	Fixed length (0–255)	Codes, short fixed strings (e.g., PIN, state code)
VARCHAR(n)	Variable length (0–65,535)	Names, email, addresses
TEXT	65,535	Articles, product descriptions
MEDIUMTEXT	16 million	Long documents
LONGTEXT	4 billion	Very large reports
ENUM('value1','value2',...)	One value from predefined list	Gender, category
SET('v1','v2',...)	Multiple values from list	Skills, tags

Example:

```
CREATE TABLE students (
    roll_no INT PRIMARY KEY,
    name VARCHAR(50),
    gender ENUM('Male', 'Female', 'Other'),
    skills SET('C','Java','Python','SQL')
);
```

✅ Saves space and restricts invalid entries.

4. Date and Time Data Types

Store time-related information.

Type	Format	Use Case
DATE	YYYY-MM-DD	Birthdate, joining date
TIME	HH:MM:SS	Time of entry
DATETIME	YYYY-MM-DD HH:MM:SS	Full timestamp
TIMESTAMP	YYYY-MM-DD HH:MM:SS	Auto-updates on changes
YEAR	YYYY	Year-only values

Example:

```
CREATE TABLE orders (
    order_id INT PRIMARY KEY,
    order_date DATE,
    delivery_time TIME,
    last_updated TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP
);
```

✅ Automatically updates last_updated when row data changes.

5. Boolean Data Type

MySQL doesn't have a strict BOOLEAN type —

Internally, it is stored as **TINYINT(1)**, where:

- 0 → False
- 1 → True

Example:

```
CREATE TABLE tasks (
    task_id INT PRIMARY KEY,
    task_name VARCHAR(50),
    completed BOOLEAN
);
```

✅ Equivalent to:

completed TINYINT(1)

6. Choosing the Right Data Type

Choosing the correct data type improves:

- **Performance** → smaller size = faster queries.
- **Storage** → less memory use.
- **Accuracy** → avoids truncation or rounding issues.
- **Data Validation** → prevents invalid data.

Guidelines

Data Type	Choose When...
INT	Counting records, IDs, numeric keys
DECIMAL	Financial or currency data
VARCHAR	Text length varies
CHAR	Fixed-length codes (e.g., country code)
DATE	Only date is needed
DATETIME	Both date and time are required
BOOLEAN	True/False flags
ENUM / SET	Restricted options or categories

Example Table Design

```
CREATE TABLE employee (
    emp_id INT PRIMARY KEY AUTO_INCREMENT,
    emp_name VARCHAR(50),
    gender ENUM('Male','Female','Other'),
    salary DECIMAL(10,2),
    date_of_join DATE,
    is_active BOOLEAN DEFAULT 1
);
```

✅ This schema balances **accuracy**, **memory efficiency**, and **data validation**.

7. Checking Data Types of a Table

After creating a table, you can verify its structure:

DESCRIBE employee;

Output:

Field	Type	Null	Key	Default	Extra
emp_id	int	NO	PRI	NULL	auto_increment
emp_name	varchar(50)	YES		NULL	
gender	enum('Male','Female','Other')	YES		NULL	
salary	decimal(10,2)	YES		NULL	
date_of_join	date	YES		NULL	
is_active	tinyint(1)	YES		1	

8. Common Data Type Mistakes

Mistake	Issue	Better Alternative
Using TEXT for short names	Wastes space	Use VARCHAR(50)
Using FLOAT for money	Rounding errors	Use DECIMAL(10,2)
Using CHAR for long text	Fixed space wasted	Use VARCHAR
Using large INT unnecessarily	Extra storage	Choose smallest type (e.g., SMALLINT)

9. Hands-On Practice (MySQL)

Task 1: Create Schema

```
CREATE DATABASE datatype_practice;
USE datatype_practice;
```

Task 2: Create Table

```
CREATE TABLE book (
    book_id INT PRIMARY KEY AUTO_INCREMENT,
    title VARCHAR(100),
    author VARCHAR(50),
```

```
    price DECIMAL(8,2),
    publish_date DATE,
    category ENUM('Fiction','Non-Fiction','Education','Comics'),
    available BOOLEAN DEFAULT 1
);
```

Task 3: Insert and Query

```
INSERT INTO book (title, author, price, publish_date, category)
VALUES ('SQL Mastery','Kavya D',499.99,'2025-06-15','Education');

SELECT * FROM book;
```

✅ Displays structured, well-typed data.

Checkpoint Review

1. Differentiate between CHAR and VARCHAR.
2. Which data type would you use for currency and why?
3. What is the internal representation of BOOLEAN in MySQL?
4. Write SQL to create a table movies with id, title, release_date, and rating.
5. Why is choosing the correct data type important for performance?

Reflection Prompt

Imagine designing a student database.
What data types would you use for roll number, name, date of birth, CGPA, and department?
Explain your choices briefly.

Summary

- Data types define what kind of data a column can store.
- **Numeric**, **String**, **Date/Time**, and **Boolean** are primary categories.
- Use **appropriate precision and length** for each column.
- Proper data typing ensures **efficiency**, **accuracy**, and **validation**.

4.3 Creating Databases & Tables

Subtopics

- Creating Databases
- Selecting and Using Databases

- Creating Tables with Constraints
- Viewing and Modifying Table Structures
- Best Practices for Table Design

Learning Outcome

After completing this topic, you will be able to:

- Create and select databases in MySQL.
- Define tables with appropriate columns and constraints.
- Modify existing table structures safely.
- Follow naming and design best practices for scalable databases.

1. Concept Overview

In MySQL, a **database (schema)** is a logical container that stores all tables, views, procedures, and other objects.
Each **table** represents an entity — with **columns** as attributes and **rows** as records.
Creating tables correctly is the **first real step in database design**.

2. Creating a Database

Syntax

```
CREATE DATABASE database_name;
```

Example

```
CREATE DATABASE college_db;
```

✅ Creates a new database named college_db.

View All Databases

```
SHOW DATABASES;
```

Select a Database to Use

```
USE college_db;
```

✅ Switches the working context to college_db.

3. Creating Tables

Tables are created inside a database to store structured data.

Syntax

```
CREATE TABLE table_name (
    column_name data_type [constraint],
    ...
);
```

Example: Student Table

```
CREATE TABLE student (
    roll_no INT PRIMARY KEY,
    name VARCHAR(50) NOT NULL,
    department VARCHAR(30),
    gender ENUM('Male','Female','Other'),
    mark DECIMAL(5,2),
    city VARCHAR(50) DEFAULT 'Chennai'
);
```

✅ Creates a student table with appropriate data types and constraints.

Explanation

Column	Type	Constraint	Purpose
roll_no	INT	PRIMARY KEY	Unique student ID
name	VARCHAR(50)	NOT NULL	Must have a name
department	VARCHAR(30)	—	Optional
gender	ENUM	Restrict to fixed values	
mark	DECIMAL(5,2)	—	Accurate marks storage
city	VARCHAR(50)	DEFAULT 'Chennai'	Default value if not given

Insert Sample Data

```
INSERT INTO student (roll_no, name, department, gender, mark)
VALUES
(101, 'Kavya', 'CSE', 'Female', 88.50),
(102, 'Arjun', 'IT', 'Male', 77.30),
(103, 'Meera', 'ECE', 'Female', 91.20);
```

View Data

```
SELECT * FROM student;
```

Output:

roll_no	name	department	gender	mark	city
101	Kavya	CSE	Female	88.50	Chennai
102	Arjun	IT	Male	77.30	Chennai
103	Meera	ECE	Female	91.20	Chennai

4. Applying Constraints

Constraints maintain data **accuracy** and **integrity**.

We can define them **during** or **after** table creation.

Constraint	Purpose	Example
PRIMARY KEY	Unique row identifier	roll_no INT PRIMARY KEY
NOT NULL	Prevents NULL values	name VARCHAR(50) NOT NULL
UNIQUE	Prevents duplicate values	email VARCHAR(100) UNIQUE
DEFAULT	Sets a default value	city VARCHAR(50) DEFAULT 'Chennai'
CHECK	Validates condition	CHECK (mark >= 0 AND mark <= 100)
FOREIGN KEY	Enforces relationships	FOREIGN KEY (dept_id) REFERENCES department(dept_id)

Adding Constraints After Creation

```
ALTER TABLE student
ADD CONSTRAINT check_mark CHECK (mark BETWEEN 0 AND 100);

ALTER TABLE student
ADD UNIQUE (name);
```

Dropping Constraints

```
ALTER TABLE student DROP INDEX name;
ALTER TABLE student DROP CHECK check_mark;
```

5. Viewing Table Structure

To check how your table is designed:

DESCRIBE student;

Output:

Field	Type	Null	Key	Default	Extra
roll_no	int	NO	PRI	NULL	
name	varchar(50)	NO		NULL	
department	varchar(30)	YES		NULL	
gender	enum('Male','Female','Other')	YES		NULL	
mark	decimal(5,2)	YES		NULL	
city	varchar(50)	YES		Chennai	

Show Table List

```
SHOW TABLES;
```

Describe Table Structure (Alternative)

```
SHOW COLUMNS FROM student;
```

6. Modifying Table Structures

You can change table structure using the ALTER TABLE command.

Add Column

```
ALTER TABLE student ADD email VARCHAR(100);
```

Modify Column

```
ALTER TABLE student MODIFY mark DECIMAL(6,2);
```

Rename Column

```
ALTER TABLE student RENAME COLUMN department TO dept;
```

Drop Column

```
ALTER TABLE student DROP COLUMN city;
```

7. Deleting Tables and Databases

Delete Table

```
DROP TABLE student;
```

Delete Database

```
DROP DATABASE college_db;
```

⚠️ **Warning:** This permanently deletes data — use carefully.

8. Best Practices for Database Design

✅ **Naming Conventions**

- Use lowercase names (student, employee_salary)
- Avoid spaces and special characters
- Use meaningful names (dept_id instead of id1)

✅ **Data Types**

- Choose minimal data type size for performance
- Use DECIMAL for currency, not FLOAT
- Use DATE or DATETIME for timestamps

✅ **Constraints**

- Always define PRIMARY KEY
- Use FOREIGN KEY for relationships
- Apply CHECK to validate data ranges

✅ **Documentation**

- Add comments for future reference

```
CREATE TABLE employee (
    emp_id INT PRIMARY KEY COMMENT 'Unique employee ID',
    emp_name VARCHAR(50) COMMENT 'Full name of employee'
);
```

9. Hands-On Practice (MySQL)

- **Create Database**

```
CREATE DATABASE college_management;
USE college_management;
```

- **Create Department Table**

```
CREATE TABLE department (
    dept_id CHAR(3) PRIMARY KEY,
    dept_name VARCHAR(50) UNIQUE
);
```

- **Create Student Table**

```
CREATE TABLE student (
    roll_no INT PRIMARY KEY,
    name VARCHAR(50) NOT NULL,
    dept_id CHAR(3),
    gender ENUM('Male','Female','Other'),
    mark DECIMAL(5,2) CHECK (mark BETWEEN 0 AND 100),
    FOREIGN KEY (dept_id) REFERENCES department(dept_id)
);
```

- **Insert Data**

```
INSERT INTO department VALUES ('D01','CSE'), ('D02','IT');
INSERT INTO student VALUES (101,'Kavya','D01','Female',89.5);
```

- **Query**

```
SELECT s.name, d.dept_name
FROM student s
JOIN department d ON s.dept_id = d.dept_id;
```

✅ Displays joined data from both tables.

Checkpoint Review

1. What is the difference between a database and a table?
2. Write SQL commands to create and use a database named company.
3. How do you add a column to an existing table?
4. What does DESCRIBE do in MySQL?
5. Create a table employee with emp_id, name, salary, and dept_id (foreign key).

Reflection Prompt

If you were designing a **hospital management database**, what tables and columns would you include?
How would you ensure data integrity using constraints?

Summary

- CREATE DATABASE creates logical storage; CREATE TABLE defines structure.
- Use **constraints** to maintain accuracy and relationships.
- ALTER TABLE modifies structure; DESCRIBE displays schema details.
- Follow naming, typing, and constraint best practices for scalable database design.

4.4 Foreign Keys & Relationships

Subtopics

- Concept of Relationships
- Primary Key vs Foreign Key
- Referential Integrity
- Creating Relationships between Tables
- ON DELETE and ON UPDATE Actions
- Practical Example with Cascading

Learning Outcome

After completing this topic, you will be able to:

- Define and apply **foreign keys** to link tables.
- Understand the importance of **referential integrity**.
- Implement cascading updates and deletes using ON DELETE and ON UPDATE.
- Design normalized, relationship-based database schemas.

1. Concept of Relationships

In a relational database, data is stored across multiple tables to **avoid redundancy**.
A **relationship** connects these tables logically using **keys**.

Example:
A *Student* belongs to a *Department*.
So, student.dept_id should refer to department.dept_id.

Types of Relationships

Type	Description	Example
One-to-One (1:1)	Each record in one table relates to only one record in another.	Aadhar ↔ Person
One-to-Many (1:N)	One record in parent table relates to many in child.	Department ↔ Students
Many-to-Many (M:N)	Many records relate on both sides (via junction table).	Students ↔ Courses

2. Primary Key vs Foreign Key

Aspect	Primary Key	Foreign Key
Purpose	Uniquely identifies each record in a table.	Refers to primary key of another table.
Uniqueness	Must be unique and not null.	Can repeat and may allow nulls.
Location	Defined in parent table.	Defined in child table.
Integrity	Ensures entity integrity.	Ensures referential integrity.

Example Relationship

```
CREATE TABLE department (
    dept_id CHAR(3) PRIMARY KEY,
    dept_name VARCHAR(50)
);

CREATE TABLE student (
    roll_no INT PRIMARY KEY,
    name VARCHAR(50),
    dept_id CHAR(3),
    FOREIGN KEY (dept_id) REFERENCES department(dept_id)
);
```

✅ Here,

- department.dept_id → **Primary Key** (Parent Table)
- student.dept_id → **Foreign Key** (Child Table)

3. Referential Integrity

Referential Integrity ensures that relationships between tables remain valid.
It prevents:

- Inserting a student with a dept_id that doesn't exist.
- Deleting a department that still has students assigned.

✅ Maintained through **Foreign Key Constraints**.

Example – Prevent Invalid Insertion

INSERT INTO student VALUES (104, 'Rahul', 'D05');

❌ Error: Cannot add or update a child row — a foreign key constraint fails.

(D05 does not exist in the parent table department)

4. Cascading Actions

MySQL supports automatic updates or deletions of child rows when parent records change.

Action	Description
ON DELETE CASCADE	Deletes child records when the parent is deleted.
ON UPDATE CASCADE	Updates foreign key values automatically if the parent key changes.
ON DELETE SET NULL	Sets child's foreign key to NULL when parent is deleted.
ON UPDATE SET NULL	Sets foreign key to NULL when parent key changes.
ON DELETE RESTRICT	Prevents deletion if related child records exist.
ON UPDATE RESTRICT	Prevents update if child records exist.

Syntax

```
FOREIGN KEY (column_name)
REFERENCES parent_table(column_name)
ON DELETE action
ON UPDATE action;
```

5. Example: Using ON DELETE and ON UPDATE

Step 1: Create Database

```
CREATE DATABASE relationship_demo;
USE relationship_demo;
```

Step 2: Create Parent Table

```
CREATE TABLE department (
    dept_id CHAR(3) PRIMARY KEY,
    dept_name VARCHAR(50)
);
```

Step 3: Create Child Table with Cascading Actions

```
CREATE TABLE student (
    roll_no INT PRIMARY KEY,
    name VARCHAR(50),
    dept_id CHAR(3),
    FOREIGN KEY (dept_id)
        REFERENCES department(dept_id)
        ON DELETE CASCADE
        ON UPDATE CASCADE
);
```

Step 4: Insert Data

```
INSERT INTO department VALUES
('D01', 'CSE'),
('D02', 'IT');

INSERT INTO student VALUES
(101, 'Kavya', 'D01'),
(102, 'Arjun', 'D02');
```

Step 5: Test ON DELETE CASCADE

```
DELETE FROM department WHERE dept_id = 'D02';
```

✅ Automatically deletes student Arjun (D02) from student table.

Step 6: Test ON UPDATE CASCADE

```
UPDATE department SET dept_id = 'D03' WHERE dept_id = 'D01';
SELECT * FROM student;
```

✅ Student Kavya's dept_id updates automatically to D03.

6. Many-to-Many Relationship Example

A **many-to-many** relationship requires a *junction (link)* table.

Example: Students ↔ Courses

```
CREATE TABLE course (
    course_id CHAR(3) PRIMARY KEY,
    course_name VARCHAR(50)
);

CREATE TABLE student_course (
    roll_no INT,
    course_id CHAR(3),
    PRIMARY KEY (roll_no, course_id),
    FOREIGN KEY (roll_no) REFERENCES student(roll_no) ON DELETE CASCADE,
    FOREIGN KEY (course_id) REFERENCES course(course_id) ON DELETE
CASCADE
);
```

✅ If a student or course is deleted, related enrollments are automatically removed.

7. Verifying Relationships

To check the foreign key definitions:

```
SHOW CREATE TABLE student;
```

To view all foreign keys in the database:

```
SELECT TABLE_NAME, CONSTRAINT_NAME, REFERENCED_TABLE_NAME
FROM INFORMATION_SCHEMA.KEY_COLUMN_USAGE
WHERE TABLE_SCHEMA = 'relationship_demo';
```

8. Modifying and Dropping Foreign Keys

Add a Foreign Key (after table creation)

```
ALTER TABLE student
ADD CONSTRAINT fk_student_dept
```

```
FOREIGN KEY (dept_id) REFERENCES department(dept_id)
ON DELETE CASCADE;
```

Drop a Foreign Key

```
ALTER TABLE student DROP FOREIGN KEY fk_student_dept;
```

9. Best Practices for Relationships

✅ Always index primary and foreign keys for performance.

✅ Use meaningful key names (fk_student_dept, not fk1).

✅ Use CASCADE carefully — avoid accidental data loss.

✅ Normalize data — store repeating information in separate tables.

✅ Use consistent naming (dept_id in both tables).

10. Hands-On Practice (MySQL)

1. **Create Schema**

```
CREATE DATABASE foreignkey_practice;
USE foreignkey_practice;
```

2. **Create Parent Table**

```
CREATE TABLE department (
    dept_id CHAR(3) PRIMARY KEY,
    dept_name VARCHAR(50)
);
```

3. **Create Child Table**

```
CREATE TABLE employee (
    emp_id INT PRIMARY KEY AUTO_INCREMENT,
    emp_name VARCHAR(50),
    dept_id CHAR(3),
    FOREIGN KEY (dept_id) REFERENCES department(dept_id)
        ON DELETE SET NULL
        ON UPDATE CASCADE
);
```

4. **Insert Data and Test**

```
INSERT INTO department VALUES ('D01', 'HR'), ('D02', 'Finance');
INSERT INTO employee (emp_name, dept_id) VALUES ('Meera', 'D01'),
('Arjun', 'D02');
DELETE FROM department WHERE dept_id='D01';
SELECT * FROM employee;  -- Meera's dept_id becomes NULL
```

Checkpoint Review

1. What is a foreign key?
2. How is a foreign key different from a primary key?
3. What does ON DELETE CASCADE do?
4. Write SQL to create a student–department relationship with cascading delete.
5. Why are foreign keys important for data integrity?

Reflection Prompt

Imagine a **Library Management System** where each book belongs to a category and is issued to members.
How would you use foreign keys to maintain relationships between **books**, **members**, and **borrowings**?

Summary

- **Foreign keys** connect tables and enforce relational integrity.
- **Parent Table** → has primary key.
- **Child Table** → references parent key.
- **Cascading actions** (ON DELETE, ON UPDATE) automate relational maintenance.
- Relationships can be **1:1**, **1:N**, or **M:N** using junction tables.
- Proper foreign key design ensures **consistency**, **accuracy**, and **referential control** in databases.

4.5 Stored Procedures & Functions

Subtopics

- Concept and Need for Stored Procedures
- Syntax and Execution
- Parameters: IN, OUT, INOUT

- User-Defined Functions (UDFs)
- Practical Real-life Examples

Learning Outcome

After completing this topic, you will be able to:

- Understand the concept and benefits of stored procedures and functions.
- Create, execute, and manage stored procedures in MySQL.
- Pass parameters using IN, OUT, and INOUT.
- Write and use reusable user-defined functions for calculations.

1. What is a Stored Procedure?

A **Stored Procedure (SP)** is a **precompiled SQL block** stored in the database.
It contains one or more SQL statements that can be executed repeatedly with a single call.

Analogy:
Instead of writing the same set of SQL commands again and again, you save them as a *procedure* and just call it by name.

Syntax

```
DELIMITER $$

CREATE PROCEDURE procedure_name (parameters)
BEGIN
   -- SQL statements
END $$

DELIMITER ;
```

Execute Procedure

CALL procedure_name(arguments);

2. Why Use Stored Procedures?

Advantage	Description
Reusability	Execute logic multiple times without rewriting SQL.
Security	Hide internal SQL from users; control permissions.
Performance	Precompiled → faster execution.
Maintainability	Easier to manage business logic centrally.

Modularity	Code is structured and organized.

3. Example 1: Simple Stored Procedure

Create Database

```
CREATE DATABASE sp_demo;
USE sp_demo;
```

Create Table

```
CREATE TABLE employee (
    emp_id INT PRIMARY KEY AUTO_INCREMENT,
    emp_name VARCHAR(50),
    salary DECIMAL(10,2),
    dept VARCHAR(30)
);
```

Insert Procedure

```
DELIMITER $$

CREATE PROCEDURE insert_employee(
    IN name_in VARCHAR(50),
    IN salary_in DECIMAL(10,2),
    IN dept_in VARCHAR(30)
)
BEGIN
    INSERT INTO employee (emp_name, salary, dept)
    VALUES (name_in, salary_in, dept_in);
END $$

DELIMITER ;
```

Execute Procedure

```
CALL insert_employee('Kavya', 65000, 'IT');
CALL insert_employee('Arjun', 55000, 'HR');
SELECT * FROM employee;
```

✅ Inserts records using the stored procedure.

4. Parameters in Stored Procedures

IN	Input only	Passes data *into* the procedure.
OUT	Output only	Returns data *from* the procedure.
INOUT	Both input & output	Passes data both ways.

Example 2: Using OUT Parameter

```
DELIMITER $$

CREATE PROCEDURE get_salary(
    IN emp_name_in VARCHAR(50),
    OUT emp_salary_out DECIMAL(10,2)
)
BEGIN
    SELECT salary INTO emp_salary_out
    FROM employee
    WHERE emp_name = emp_name_in;
END $$

DELIMITER ;
```

Call Procedure:

```
CALL get_salary('Kavya', @sal);
SELECT @sal AS Employee_Salary;
```

✅ Returns the salary for the given employee.

Example 3: Using INOUT Parameter

```
DELIMITER $$

CREATE PROCEDURE increment_salary(
    INOUT emp_sal DECIMAL(10,2)
)
BEGIN
    SET emp_sal = emp_sal + 5000;
END $$

DELIMITER ;
```

Execute:

```
SET @mysal = 45000;
CALL increment_salary(@mysal);
SELECT @mysal AS Updated_Salary;
```

✅ Updates the value of the variable and returns it.

5. Example 4: Conditional Logic in Stored Procedure

Stored procedures can use control statements like IF, CASE, and LOOP.

```
DELIMITER $$

CREATE PROCEDURE adjust_salary(IN emp_name_in VARCHAR(50))
BEGIN
    DECLARE emp_sal DECIMAL(10,2);
    SELECT salary INTO emp_sal FROM employee WHERE emp_name =
emp_name_in;

    IF emp_sal < 60000 THEN
        UPDATE employee SET salary = emp_sal + 5000 WHERE emp_name =
emp_name_in;
    ELSE
        UPDATE employee SET salary = emp_sal + 2000 WHERE emp_name =
emp_name_in;
    END IF;
END $$

DELIMITER ;
```

Call:

```
CALL adjust_salary('Arjun');
SELECT * FROM employee;
```

✅ Increases Arjun's salary conditionally.

6. Example 5: Looping with Cursor in Stored Procedure

Cursors can be used inside procedures for row-by-row operations.

```
DELIMITER $$

CREATE PROCEDURE increase_all_salaries()
BEGIN
```

```
    DECLARE done INT DEFAULT 0;
    DECLARE e_id INT;
    DECLARE e_salary DECIMAL(10,2);
    DECLARE emp_cursor CURSOR FOR SELECT emp_id, salary FROM employee;
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;

    OPEN emp_cursor;
    read_loop: LOOP
        FETCH emp_cursor INTO e_id, e_salary;
        IF done THEN
            LEAVE read_loop;
        END IF;
        UPDATE employee SET salary = e_salary * 1.05 WHERE emp_id = e_id;
    END LOOP;
    CLOSE emp_cursor;
END $$

DELIMITER ;
```

Call:

```
CALL increase_all_salaries();
SELECT * FROM employee;
```

✅ Increases all salaries by 5%.

7. Managing Stored Procedures

View All Procedures

```
SHOW PROCEDURE STATUS WHERE Db = 'sp_demo';
```

View Procedure Code

```
SHOW CREATE PROCEDURE insert_employee;
```

Drop a Procedure

```
DROP PROCEDURE insert_employee;
```

8. What is a Function in MySQL?

A **User-Defined Function (UDF)** is similar to a stored procedure but **returns a single value** and can be used directly inside SQL queries.

Syntax

```
DELIMITER $$

CREATE FUNCTION function_name (parameters)
RETURNS datatype
DETERMINISTIC
BEGIN
   -- SQL statements
   RETURN value;
END $$

DELIMITER ;
```

Key Differences: Stored Procedure vs Function

Aspect	Stored Procedure	Function
Return Type	No return value (uses OUT)	Must return a single value
Call Method	CALL procedure_name()	Used inside SELECT
Usage	Business logic, updates	Computations, formatting
DML Allowed	Yes	Limited (mostly read-only)

9. Example: Creating a Function

```
DELIMITER $$

CREATE FUNCTION get_bonus(salary DECIMAL(10,2))
RETURNS DECIMAL(10,2)
DETERMINISTIC
BEGIN
    DECLARE bonus DECIMAL(10,2);
    SET bonus = salary * 0.10;
    RETURN bonus;
END $$

DELIMITER ;
```

Use the Function:

```
SELECT emp_name, salary, get_bonus(salary) AS Bonus
FROM employee;
```

✅ Returns a computed bonus for each employee.

10. Example: Function with Conditions

```
DELIMITER $$

CREATE FUNCTION tax_deduction(salary DECIMAL(10,2))
RETURNS DECIMAL(10,2)
DETERMINISTIC
BEGIN
    DECLARE tax DECIMAL(10,2);
    IF salary <= 50000 THEN
        SET tax = salary * 0.05;
    ELSEIF salary <= 80000 THEN
        SET tax = salary * 0.10;
    ELSE
        SET tax = salary * 0.15;
    END IF;
    RETURN tax;
END $$

DELIMITER ;
```

Call Function:

```
SELECT emp_name, salary, tax_deduction(salary) AS Tax FROM employee;
```

✅ Calculates tax for each employee dynamically.

11. Best Practices

✅ Use meaningful procedure and parameter names.

✅ Add comments to explain business logic.

✅ Use DETERMINISTIC keyword when output is predictable.

✅ Avoid heavy loops — use set-based operations where possible.

✅ Test procedures before applying to production data.

12. Hands-On Practice (MySQL)

1. **Create Schema and Table**

```
CREATE DATABASE function_practice;
```

```
USE function_practice;

CREATE TABLE orders (
    order_id INT PRIMARY KEY AUTO_INCREMENT,
    amount DECIMAL(10,2)
);

INSERT INTO orders (amount) VALUES (1200),(2500),(3600);
```

2. **Create Function**

```
DELIMITER $$

CREATE FUNCTION discount(amount DECIMAL(10,2))
RETURNS DECIMAL(10,2)
DETERMINISTIC
BEGIN
    DECLARE discounted DECIMAL(10,2);
    SET discounted = amount * 0.90;
    RETURN discounted;
END $$

DELIMITER ;
```

3. **Use in Query**

```
SELECT order_id, amount, discount(amount) AS discounted_price FROM
orders;
```

✅ Demonstrates function usage within standard queries.

Checkpoint Review

1. What is a stored procedure, and why is it used?
2. Difference between IN, OUT, and INOUT parameters.
3. Write a stored procedure to calculate and update employee bonuses.
4. Create a function to return the total price with tax.
5. How is a stored procedure different from a function in MySQL?

Reflection Prompt

If you were automating salary and tax management in a payroll database,
how would you design stored procedures and functions to handle monthly payroll generation?

Summary

- **Stored Procedures** encapsulate business logic in reusable SQL blocks.
- **Parameters** allow flexibility in input and output handling.
- **Functions** return single computed values for use inside queries.
- Using SPs & Functions improves **modularity**, **security**, and **performance**.
- Both are critical for real-world database applications and backend integration.

4.6 Triggers

Subtopics

- Concept and Need for Triggers
- BEFORE vs AFTER Triggers
- Syntax & Structure
- Practical Use Cases (Audit, Validation, Auto-update)
- Trigger Management (Viewing, Dropping, Testing)

Learning Outcome

After completing this topic, you will be able to:

- Understand what triggers are and why they're used.
- Create and manage triggers in MySQL.
- Differentiate between BEFORE and AFTER triggers.
- Apply triggers to automate tasks like logging, enforcing rules, or maintaining history.

1. What is a Trigger?

A **Trigger** is a **database object** that executes **automatically** in response to specific events (INSERT, UPDATE, or DELETE) on a table.

In simple terms:
When something happens in a table → a trigger "fires" → it executes the SQL logic you define.

Analogy

Think of a trigger like a **door alarm** —
when someone opens (event), the alarm rings (trigger action).

2. Trigger Events in MySQL

Event	Description
INSERT	Trigger fires when a new row is inserted.
UPDATE	Trigger fires when a row is updated.
DELETE	Trigger fires when a row is deleted.

3. BEFORE vs AFTER Triggers

Type	When It Fires	Use Case
BEFORE Trigger	Executes *before* the event occurs.	Data validation, auto-formatting.
AFTER Trigger	Executes *after* the event occurs.	Auditing, logging, summaries.

Trigger Access Keywords

Keyword	Meaning
NEW	Refers to the new row being inserted or updated.
OLD	Refers to the existing row being updated or deleted.

4. Trigger Syntax

```
DELIMITER $$

CREATE TRIGGER trigger_name
{BEFORE | AFTER} {INSERT | UPDATE | DELETE}
ON table_name
FOR EACH ROW
BEGIN
   -- Trigger logic
END $$

DELIMITER ;
```

5. Example 1: BEFORE INSERT Trigger (Data Validation)

```
CREATE DATABASE trigger_demo;
USE trigger_demo;

CREATE TABLE employee (
    emp_id INT PRIMARY KEY AUTO_INCREMENT,
    emp_name VARCHAR(50),
    salary DECIMAL(10,2)
);
```

Trigger: Prevent Negative Salary

```
DELIMITER $$

CREATE TRIGGER check_salary
BEFORE INSERT ON employee
FOR EACH ROW
BEGIN
    IF NEW.salary < 0 THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Salary cannot be negative!';
    END IF;
END $$

DELIMITER ;
```

Test Trigger

```
INSERT INTO employee (emp_name, salary) VALUES ('Kavya', 50000);
INSERT INTO employee (emp_name, salary) VALUES ('Arjun', -20000);
```

✅ First query inserts successfully.

❌ Second query fails with: *Salary cannot be negative!*

6. Example 2: AFTER INSERT Trigger (Logging)

Step 1: Create Log Table

```
CREATE TABLE emp_log (
    log_id INT AUTO_INCREMENT PRIMARY KEY,
    emp_name VARCHAR(50),
    action VARCHAR(50),
```

```
    log_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Step 2: Create Trigger

```
DELIMITER $$

CREATE TRIGGER after_employee_insert
AFTER INSERT ON employee
FOR EACH ROW
BEGIN
    INSERT INTO emp_log (emp_name, action)
    VALUES (NEW.emp_name, 'Inserted new employee');
END $$

DELIMITER ;
```

Test

```
INSERT INTO employee (emp_name, salary) VALUES ('Meera', 60000);
SELECT * FROM emp_log;
```

Output:

log_id	emp_name	action	log_time
1	Meera	Inserted new employee	2025-10-07 12:34:56

✅ Automatically logs employee addition.

7. Example 3: AFTER UPDATE Trigger (Audit Salary Changes)

Step 1: Create Audit Table

```
CREATE TABLE salary_audit (
    audit_id INT AUTO_INCREMENT PRIMARY KEY,
    emp_id INT,
    old_salary DECIMAL(10,2),
    new_salary DECIMAL(10,2),
    changed_on TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Step 2: Create Trigger

```
DELIMITER $$
```

```
CREATE TRIGGER after_salary_update
AFTER UPDATE ON employee
FOR EACH ROW
BEGIN
    IF OLD.salary <> NEW.salary THEN
        INSERT INTO salary_audit (emp_id, old_salary, new_salary)
        VALUES (OLD.emp_id, OLD.salary, NEW.salary);
    END IF;
END $$

DELIMITER ;
```

Test

```
UPDATE employee SET salary = 65000 WHERE emp_name = 'Meera';
SELECT * FROM salary_audit;
```

✅ Automatically logs old and new salary whenever an update happens.

8. Example 4: AFTER DELETE Trigger (Track Deletions)

```
DELIMITER $$

CREATE TRIGGER after_employee_delete
AFTER DELETE ON employee
FOR EACH ROW
BEGIN
    INSERT INTO emp_log (emp_name, action)
    VALUES (OLD.emp_name, 'Deleted employee');
END $$

DELIMITER ;
```

Test:

```
DELETE FROM employee WHERE emp_name = 'Kavya';
SELECT * FROM emp_log;
```

✅ Logs the deletion automatically.

9. Viewing and Managing Triggers

Show All Triggers

```
SHOW TRIGGERS;
```

Show Trigger Definition

```
SHOW CREATE TRIGGER after_salary_update;
```

Drop a Trigger

```
DROP TRIGGER after_employee_insert;
```

10. Real-World Use Cases

Scenario	Trigger Example
Audit Trail	Log updates/deletes to a history table.
Data Validation	Prevent invalid data (e.g., negative values, missing references).
Auto Calculations	Auto-update derived columns or totals.
Sync Tables	Mirror changes to backup tables.
Security Enforcement	Prevent unauthorized data changes.

11. Best Practices

✅ Use triggers for **row-level automation**, not business logic.

✅ Avoid complex triggers — they may impact performance.

✅ Don’t chain multiple triggers on the same event unnecessarily.

✅ Use meaningful names like before_insert_validation or after_update_audit.

✅ Keep audit tables separate from main transactional tables.

12. Hands-On Practice (MySQL)

1. **Create Database**

```
CREATE DATABASE trigger_practice;
USE trigger_practice;
```

2. **Create Tables**

```
CREATE TABLE student (
    roll_no INT PRIMARY KEY,
    name VARCHAR(50),
    marks DECIMAL(5,2)
);

CREATE TABLE student_audit (
    audit_id INT AUTO_INCREMENT PRIMARY KEY,
    roll_no INT,
    action VARCHAR(50),
    old_marks DECIMAL(5,2),
    new_marks DECIMAL(5,2),
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

3. **Create Trigger**

```
DELIMITER $$

CREATE TRIGGER after_student_update
AFTER UPDATE ON student
FOR EACH ROW
BEGIN
    INSERT INTO student_audit (roll_no, action, old_marks, new_marks)
    VALUES (OLD.roll_no, 'Updated Marks', OLD.marks, NEW.marks);
END $$

DELIMITER ;
```

4. **Test**

```
INSERT INTO student VALUES (101, 'Kavya', 78.50);
UPDATE student SET marks = 89.75 WHERE roll_no = 101;
SELECT * FROM student_audit;
```

✅ Automatically records mark change in audit table.

Checkpoint Review

1. What is a trigger and when is it executed?
2. Difference between BEFORE and AFTER triggers.
3. Write a trigger to log deleted employee data.
4. What are OLD and NEW keywords used for?
5. List common use cases of triggers in real-world applications.

Reflection Prompt

Imagine you're designing a **Library Management System** where you need to log every time a book is borrowed or returned.
How would you use triggers to record this automatically?

Summary

- A **Trigger** executes automatically when a defined event occurs on a table.
- BEFORE triggers are used for validation; AFTER for logging and auditing.
- NEW and OLD represent row values before/after the event.
- Common uses: **validation**, **logging**, **auto-updates**, **audits**.
- Triggers improve automation but must be used carefully for performance and debugging.

4.7 Backup & Restore

Subtopics

- Importance of Backup & Restore
- Backup using MySQL Workbench
- Backup using mysqldump (Command-Line)
- Restoring Databases
- Best Practices for Data Safety

Learning Outcome

After completing this topic, you will be able to:

- Perform full and partial database backups using both Workbench and CLI.
- Restore databases safely from .sql dump files.
- Understand backup file structure and formats.
- Apply practical strategies to prevent data loss.

1. Concept Overview

Backup means creating a copy of your database so that you can restore it in case of data loss, corruption, or migration.
Restore means loading data back into MySQL from that backup file.

Why Backups Are Important

✅ Protects against **accidental deletions or updates**

✅ Essential during **server migrations or version upgrades**

✅ Enables **data recovery** after crashes or hacks

✅ Supports **testing, training, and cloning** environments

2. Backup Types in MySQL

Type	Description	Example
Full Backup	Complete copy of the entire database	All tables, triggers, procedures
Partial Backup	Specific tables or schema	Only student and department tables
Logical Backup	SQL statements (portable .sql file)	Using mysqldump
Physical Backup	Copying data files directly	Requires server access (for admins)

3. Backup Using MySQL Workbench (GUI Method)

Best for beginners and quick exports.

Step-by-Step Process

1. Open **MySQL Workbench** → connect to your server.
2. Go to **Server** → **Data Export**.
3. Select the **database** you want to back up.
4. Choose:
    - **Export to Self-Contained File (.sql)**
    - Example: D:\backups\college_db_backup.sql
5. Click **Start Export**.

✅ Creates a .sql file containing all tables, data, triggers, and stored procedures.

Backup Options

- **Dump Structure and Data** – full export
- **Dump Structure Only** – schema only (no data)
- **Dump Data Only** – just the records
- **Include CREATE DATABASE** – adds a database creation line

Output Example (Inside the .sql file)

```
-- MySQL dump 10.13  Distrib 8.0.36
CREATE DATABASE IF NOT EXISTS `college_db`;
USE `college_db`;

CREATE TABLE `student` (
  `roll_no` int NOT NULL,
  `name` varchar(50) DEFAULT NULL,
  PRIMARY KEY (`roll_no`)
);

INSERT INTO `student` VALUES (101,'Kavya'),(102,'Arjun');
```

4. Restore Using MySQL Workbench

Step-by-Step Process

1. Open **MySQL Workbench → Server → Data Import**.
2. Choose **Import from Self-Contained File**.
3. Select your .sql backup file (e.g., college_db_backup.sql).
4. Choose:
    - **Default Target Schema** → create a new database or select existing one.
5. Click **Start Import**.

✅ Restores all tables, data, and objects automatically.

Verification

After import:

```
SHOW DATABASES;
USE college_db;
SHOW TABLES;
SELECT * FROM student;
```

✅ Confirms the backup was restored successfully.

5. Backup Using Command Line (mysqldump Tool)

mysqldump is MySQL's built-in command-line utility for creating logical backups.

Syntax

```
mysqldump -u username -p database_name > backup_file.sql
```

Example:

```
mysqldump -u root -p college_db > D:\backups\college_db_backup.sql
```

✅ Prompts for password, then exports college_db to a .sql file.

Backup Multiple Databases

```
mysqldump -u root -p --databases db1 db2 > all_databases_backup.sql
```

Backup All Databases

```
mysqldump -u root -p --all-databases > full_server_backup.sql
```

Backup Specific Table

```
mysqldump -u root -p college_db student > student_backup.sql
```

✅ Exports only the student table.

6. Restore Using Command Line

Syntax

```
mysql -u username -p database_name < backup_file.sql
```

Example:

```
mysql -u root -p college_db < D:\backups\college_db_backup.sql
```

✅ Restores database contents from the backup file.

If Database Doesn't Exist

```
mysql -u root -p < college_db_backup.sql
```

The dump file usually includes a CREATE DATABASE statement.

7. Partial Backup and Restore

You can selectively export/import specific tables.

Backup:

```
mysqldump -u root -p college_db student department > partial_backup.sql
```

Restore:

```
mysql -u root -p college_db < partial_backup.sql
```

✅ Useful for testing or transferring a few tables.

8. Scheduling Regular Backups (Automation)

For production systems, use OS-level schedulers to automate backups.

Windows Task Scheduler Example

- Create a .bat file:

```
mysqldump -u root -pYourPassword --all-databases > "D:
\backups\backup_%DATE%.sql"
```

- Schedule it to run **daily at midnight**.

Linux Cron Job Example

```
0 0 * * * mysqldump -u root -pYourPassword --all-databases > /backups/
backup_$(date +\%F).sql
```

✅ Automatically maintains daily backups with date tags.

9. Best Practices for Backup & Restore

✅ **Always verify backup files** by checking size or restoring to a test database.

✅ Use **naming conventions** (e.g., backup_2025_10_07.sql).

✅ Store backups in **secure, offsite, and versioned locations** (e.g., Google Drive, AWS S3).

✅ Schedule **daily incremental** and **weekly full backups** for production databases.

✅ Use **compression (zip/tar)** for large backups.

✅ Always **test restoration** before relying on backups.

10. Hands-On Practice (MySQL)

1. **Create Database**

```
CREATE DATABASE backup_practice;
USE backup_practice;

CREATE TABLE products (
    id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(50),
    price DECIMAL(8,2)
);

INSERT INTO products (name, price)
VALUES ('Laptop', 55000), ('Mouse', 750), ('Keyboard', 1250);
```

2. **Backup Command (CLI)**

```
mysqldump -u root -p backup_practice > D:\backups\backup_practice.sql
```

3. **Drop and Restore**

```
DROP DATABASE backup_practice;

mysql -u root -p < D:\backups\backup_practice.sql
```

4. **Verify**

```
SHOW DATABASES;
USE backup_practice;
SELECT * FROM products;
```

✅ Database successfully restored from backup.

Checkpoint Review

1. What's the difference between **full** and **partial** backups?
2. How do you back up all databases using mysqldump?
3. Explain the role of the .sql file in MySQL backups.
4. Which command restores a database from a dump file?
5. Why should you test restoration after every backup?

Reflection Prompt

Imagine you manage a **university management system**.
How would you design a **backup strategy** that ensures no student or exam data is ever lost — even during system crashes or updates?

Summary

- **Backups** prevent data loss and aid in recovery after failures.
- **Workbench** → GUI export/import.
- **mysqldump** → Command-line tool for full and selective backups.
- **mysql** → Used for restoration.
- **Best practice**: automate, version, secure, and test backups regularly.

4.8 Mini Project: Library Management System

Objective

To design and implement a **Library Management System** in MySQL that efficiently manages **books, members, borrowings, and transactions**, integrating real-time automation using **triggers** and **stored procedures**.

1. Problem Definition

The library should:

- Store and manage book details.
- Maintain records of members who borrow or return books.
- Prevent invalid operations (like borrowing unavailable books).
- Automatically log all borrow and return events.
- Allow easy data recovery via backup.

2. Database Design Overview

Entities (Tables)

Entity	Description
Book	Contains details about books in the library.
Member	Stores library member details.
Borrowing	Tracks which member borrowed which book and when.
Audit_Log	Logs all borrow/return actions automatically via triggers.

3. Step-by-Step Implementation

Step 1: Create Database

```
CREATE DATABASE library_management;
USE library_management;
```

Step 2: Create book Table

```
CREATE TABLE book (
    book_id INT AUTO_INCREMENT PRIMARY KEY,
    title VARCHAR(100) NOT NULL,
    author VARCHAR(50),
    category VARCHAR(30),
    total_copies INT DEFAULT 1,
    available_copies INT DEFAULT 1,
    price DECIMAL(8,2)
);
```

✅ available_copies helps manage borrow/return logic.

Step 3: Create member Table

```
CREATE TABLE member (
    member_id INT AUTO_INCREMENT PRIMARY KEY,
    member_name VARCHAR(50) NOT NULL,
    email VARCHAR(100) UNIQUE,
    phone VARCHAR(15),
    membership_date DATE DEFAULT (CURRENT_DATE)
```

```
);
```

✅ Stores unique members with registration date tracking.

Step 4: Create borrowing Table

```
CREATE TABLE borrowing (
    borrow_id INT AUTO_INCREMENT PRIMARY KEY,
    member_id INT,
    book_id INT,
    borrow_date DATE DEFAULT (CURRENT_DATE),
    return_date DATE,
    status ENUM('Borrowed','Returned') DEFAULT 'Borrowed',
    FOREIGN KEY (member_id) REFERENCES member(member_id)
        ON DELETE CASCADE ON UPDATE CASCADE,
    FOREIGN KEY (book_id) REFERENCES book(book_id)
        ON DELETE CASCADE ON UPDATE CASCADE
);
```

✅ Relationships ensure referential integrity:

- Deleting a member removes their borrow history.
- Deleting a book removes related borrow records.

Step 5: Create audit_log Table

```
CREATE TABLE audit_log (
    log_id INT AUTO_INCREMENT PRIMARY KEY,
    action_type VARCHAR(20),
    member_id INT,
    book_id INT,
    action_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (member_id) REFERENCES member(member_id),
    FOREIGN KEY (book_id) REFERENCES book(book_id)
);
```

✅ Logs all borrow and return actions automatically.

4. Insert Sample Data

```
INSERT INTO book (title, author, category, total_copies,
available_copies, price)
VALUES
```

```
('DBMS Concepts', 'Elmasri', 'Education', 3, 3, 599.00),
('Let Us C', 'Yashwant Kanetkar', 'Programming', 5, 5, 399.00),
('Python Crash Course', 'Eric Matthes', 'Programming', 4, 4, 499.00);

INSERT INTO member (member_name, email, phone)
VALUES
('Kavya', 'kavya@gmail.com', '9876543210'),
('Arjun', 'arjun@gmail.com', '9123456789');
```

5. Create Stored Procedures

(a) Borrow a Book Procedure

This procedure ensures:

- Book is available before borrowing.
- Updates availability count.
- Records borrow transaction.

```
DELIMITER $$

CREATE PROCEDURE borrow_book(IN m_id INT, IN b_id INT)
BEGIN
    DECLARE copies_left INT;

    SELECT available_copies INTO copies_left
    FROM book WHERE book_id = b_id;

    IF copies_left > 0 THEN
        INSERT INTO borrowing (member_id, book_id)
        VALUES (m_id, b_id);
        UPDATE book SET available_copies = available_copies - 1
        WHERE book_id = b_id;
    ELSE
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Book not available for borrowing!';
    END IF;
END $$

DELIMITER ;
```

Test Procedure

```
CALL borrow_book(1, 2);
```

```
SELECT * FROM borrowing;
```

✅ Inserts record and decrements book stock.

(b) Return a Book Procedure

```
DELIMITER $$

CREATE PROCEDURE return_book(IN m_id INT, IN b_id INT)
BEGIN
    UPDATE borrowing
    SET return_date = CURRENT_DATE, status = 'Returned'
    WHERE member_id = m_id AND book_id = b_id AND status = 'Borrowed';

    UPDATE book
    SET available_copies = available_copies + 1
    WHERE book_id = b_id;
END $$

DELIMITER ;
```

Test Procedure

```
CALL return_book(1, 2);
SELECT * FROM borrowing;
```

✅ Updates status and restores stock.

6. Create Triggers

(a) AFTER INSERT Trigger (Log Borrow Action)

```
DELIMITER $$

CREATE TRIGGER after_borrow_insert
AFTER INSERT ON borrowing
FOR EACH ROW
BEGIN
    INSERT INTO audit_log (action_type, member_id, book_id)
    VALUES ('Borrowed', NEW.member_id, NEW.book_id);
END $$

DELIMITER ;
```

(b) AFTER UPDATE Trigger (Log Return Action)

```
DELIMITER $$

CREATE TRIGGER after_borrow_update
AFTER UPDATE ON borrowing
FOR EACH ROW
BEGIN
    IF NEW.status = 'Returned' THEN
        INSERT INTO audit_log (action_type, member_id, book_id)
        VALUES ('Returned', NEW.member_id, NEW.book_id);
    END IF;
END $$

DELIMITER ;
```

✅ Automatically logs every borrow and return activity.

Check Audit Logs

```
SELECT * FROM audit_log ORDER BY action_time DESC;
```

7. Backup the Database

Using Command Line

```
mysqldump -u root -p library_management > D:
\backups\library_management.sql
```

✅ Full backup of database, tables, stored procedures, and triggers.

Restore

```
mysql -u root -p < D:\backups\library_management.sql
```

✅ Fully restores library database with all logic intact.

8. Testing the Complete Flow

1. **Borrow a Book**

```
CALL borrow_book(2, 1);
```

2. **Return a Book**

```
CALL return_book(2, 1);
```

3. **Check Borrowing History**

```
SELECT * FROM borrowing;
```

4. **Check Logs**

```
SELECT * FROM audit_log;
```

✅ Output should show all borrow and return activities recorded with timestamps.

9. Optional Enhancements

Feature	Description
Fine Calculation	Add fine_amount column in borrowing, update via trigger when late.
View Active Borrowers	Create a view for status='Borrowed'.
Stored Function	Function to calculate book availability percentage.
Email Alerts (External Integration)	Connect via Python/Java backend to notify members.

10. ER Diagram (Textual Representation)

```
[Book] 1 ——< [Borrowing] >—— 1 [Member]
   | book_id (PK)                    | borrow_id (PK)
   | title                           | member_id (FK)
   | available_copies                | book_id (FK)
```

✅ One **book** can be borrowed by many members; one **member** can borrow many books.

Checkpoint Review

1. How are stored procedures and triggers used together in this system?
2. What happens if a member tries to borrow a book that's unavailable?
3. What is the purpose of available_copies in the book table?
4. Write SQL to view all books currently borrowed by "Kavya".
5. How can you recover the database if it gets corrupted?

Reflection Prompt

If this library expands across multiple branches, how would you modify the schema to manage branch-specific book inventories and transactions?

Summary

- **Database:** library_management
- **Core Tables:** book, member, borrowing, audit_log
- **Relationships:** 1–N between book ↔ borrowing and member ↔ borrowing
- **Stored Procedures:** automate borrowing and returning logic
- **Triggers:** auto-log every borrow/return
- **Backup:** ensures data safety and easy restoration