**MOVIE TICKET MANAGEMENT DATABSE**

21CSC205P Database Management Systems

A MINI PROJECT REPORT

*Submitted by*

**T YASHWANTH REDDY(RA2311030010317)**

**L.KOUSHIK (RA2311030010283)**

*Under the Guidance of*

## Dr.M. SIVAKUMAR

Assistant Professor, Department of Networking and Communications

*in partial fulfillment of the requirements for the degree of*

BACHELOR OF TECHNOLOGY

*in*

COMPUTER SCIENCE AND ENGINEERING

with Specialization in CYBER SECURITY****

**DEPARTMENT OF NETWORKING AND COMMUNICATIONS**

**COLLEGE OF ENGINEERING AND TECHNOLOGY**

**SRM INSTITUTE OF SCIENCE AND TECHNOLOGY**

**KATTANKULATHUR– 603 203**

**MAY 2025**



**SRM INSTITUTE OF SCIENCE AND TECHNOLOGY KATTANKULATHUR 603203**

BONAFIDE CERTIFICATE

Certified that Project report titled “**MOVIE TICKET BOOKING DATABASE**” is the bonafide work of “**T. YASHWANTH REDDY [RA2311030010317], L. KOUSHIK[RA2311030010283]”** who carried out the **21CSC205P Database Management Systems** mini project work under my supervision.

**Faculty in Charge HEAD OF THE DEPARTMENT**

Dr.M.SHIVAKUMAR Dr.M. Lakshmi

Assistant Professor Professor & Head

Department of NWC Department of NWC

SRMIST-KTR SRMIST-KTR

**ABSTRACT**

The Movie Ticket Booking System (MovieTixDB) is a comprehensive and scalable database-driven application designed to modernize the reservation and management of cinema screenings across multiplexes. This system provides a structured digital framework to manage core entities such as movies, theaters, showtimes, seats, customers, and bookings. Developed with MySQL as the backend and Python as the frontend using Tkinter for the GUI, the platform ensures seamless interaction, transactional integrity, and consistent data flow across all user modules. The database architecture follows a normalized relational schema with foreign key constraints, indexing, and stored procedures to ensure high performance and data accuracy. It supports real-time seat availability, secure booking transactions, and dynamic scheduling by theater administrators. Robust transaction management, including ACID compliance and features like write-ahead logging (WAL), rollback support, and point-in-time recovery, ensures data integrity during concurrent accesses and potential system failures. Role-based access control is enforced to differentiate between customer and admin privileges, enhancing operational security. The system also integrates receipt generation, booking history logs, and refund mechanisms. With potential for future integration of QR code-based ticketing and mobile app compatibility, MovieTixDB offers a reliable, user-centric, and scalable solution for cinema ticket booking in the digital age.

# TABLE OF CONTENTS

|  |  |  |
| --- | --- | --- |
| **Chapter No** | **Chapter Name** | **Page No** |
| **1.** | Introduction  1.1Problem understanding  1.2Identification of Entity and Relationship  1.3Construction of DB using ER model for Project | 5 |
| **2.** | Data Design  2.1 Design of Relational Schemas  2.2Creation of database tables for the project | 10 |
| **3.** | Exection of Queries  3.1 constraints  3.2 Sets  3.3 Joins  3.4 Views  3.5 Triggers  3.6 Cursor | 22 |
| **4.** | Normalization  4.1 Analyzing the pitfalls  4.2Indentifying the dependencies  4.3Applying normalizations | 34 |
| **5.** | Concurrency control and recovery mechanism  5.1 Implementation of concurrency control  5.2 Implementation of Recovery mechanism | 38 |
| **6.** | Code for the project | 42 |
| **7.** | Result and Discussion (Screen shots of the implementation with front end.)  7.1 Youtube Demonstration  7.2 Github Repostery  7.3 References | 45 |

**Problem Statement:**

The current movie ticket booking process is plagued by inefficiencies stemming from manual operations, long queues, lack of centralized scheduling, and inconsistent seat management. These shortcomings lead to booking errors, customer dissatisfaction, overbookings, and limited accessibility for users and theatre staff. A robust, secure, and interactive digital system is essential to streamline ticket reservations, enhance user experience, and optimize theatre management operations.

**Problem Description:**

1. **Manual and Paper-Based Processes:** Traditional ticket counters rely on manual ticket sales, causing long waiting times, human errors in seat allocation, and a frustrating customer experience.
2. **Lack of Real-Time Coordination:** Theatre staff often struggle to update movie schedules, seat availability, or booking status in real time, leading to double bookings or inconsistent records.
3. **Inconsistent Data Management:** Customers have minimal ability to modify or cancel bookings, choose preferred seats, or explore shows based on filters like language, genre, or time slots.
4. **Limited Transparency and Traceability:** Stakeholders have minimal visibility into the organ procurement and allocation process, leading to mistrust and potential ethical concerns.
5. **Security and Privacy Risks:** Sensitive personal and medical information of donors and recipients is vulnerable in unprotected systems, posing risks to data privacy and compliance with regulations.
6. **Poor User Experience:** Existing systems (if any) may not offer intuitive interfaces for different users (e.g., doctors, coordinators, donors), making it difficult to efficiently navigate and perform critical tasks.
7. **Limited Scalability and Integration:** Current solutions cannot scale effectively with increasing data and user demands, and often lack integration with national or regional health networks and databases.

# Chapter 1

ER-DIAGRAM FOR ORGAN DONATION & PROCUREMENT NETWORK MANAGEMENT SYSTEM:

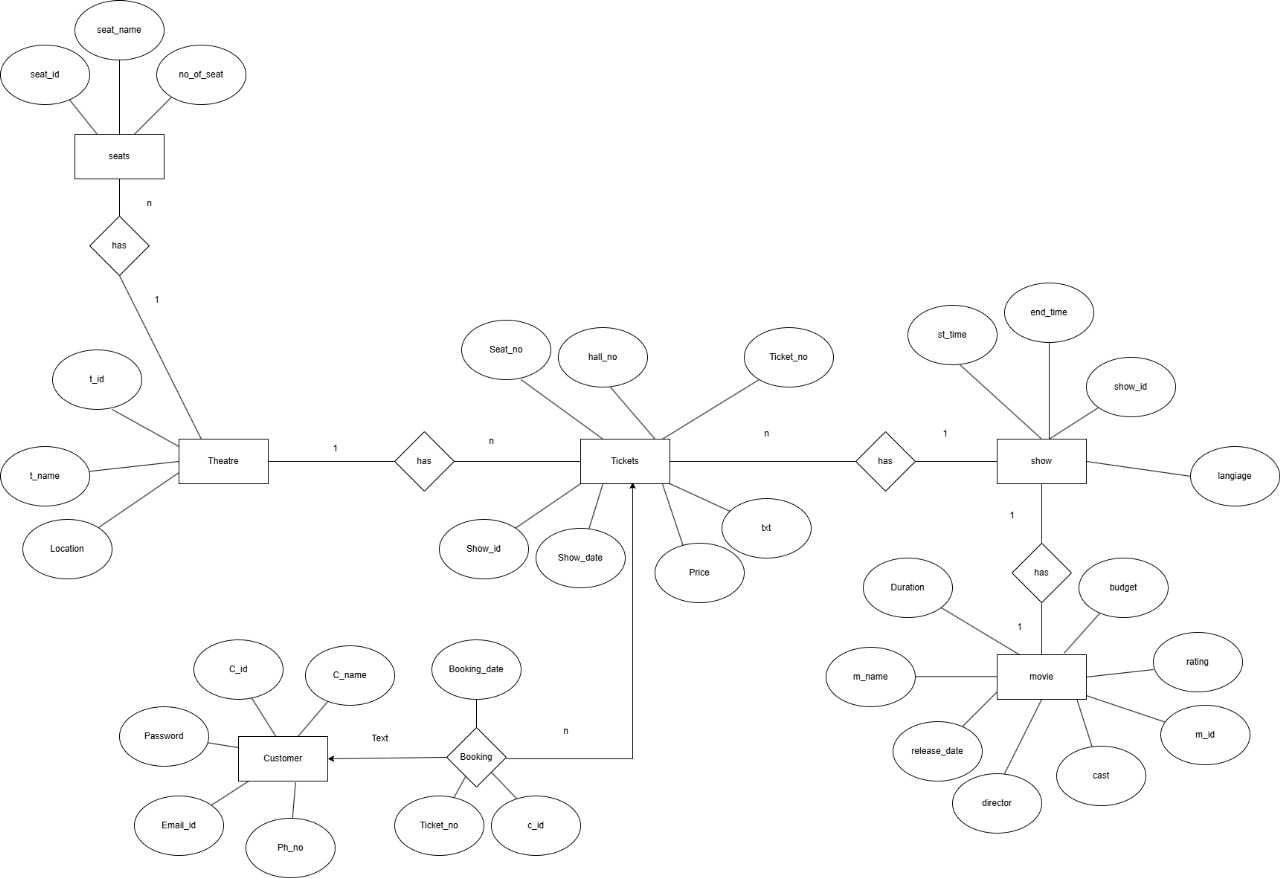


Fig:1.1 ER DIAGRAM

# 1.2Design of Relational Schemas, Creation of Database Tables for the

# Movie Ticket Booking System

T The Entity-Relationship (ER) diagram for the Movie Ticket Booking System visually represents the structured interaction between key entities such as Customers, Movies, Theatres, Shows, Tickets, and Payments. It outlines how data flows between these entities and defines the relationships and attributes that govern ticket reservations, seat selection, and transaction management. This diagram serves as a foundational blueprint for ensuring efficient booking operations, accurate seat allocation, and seamless coordination between users and theatre administrators.

**Entities and Their Roles**

1. **User**

* **Attributes:**
  + CUSTOMER\_ID (Primary Key)
  + NAME
  + EMAIL\_ID
  + MOBILE\_NO

## Relationships:

* + Books tickets through the **BOOKS** relationship
  + Linked to **Payment** for transaction tracking.
  + Can have multiple **Tickets** (One-to-Many).

## Movie

* **Attributes:**
  + MOVIE\_ID (Primary Key)
  + MOVIE\_NAME
  + GENRE
  + RELEASE\_DATE
  + DURATION

## Theatre

* **Attributes:**
  + THEATRE\_ID (Primary Key)
  + THEATRE\_NAME
  + THEATRE\_LOCATION
  + THEATRE\_CAPACITY
  + THEATRE\_EMAIL\_ID

## Relationships:

* + Hosts multiple **Shows** and **Seats** (One-to-Many).
  + Linked to **Tickets** for managing seat reservations.

## SHOW

* **Attributes:**
  + SHOW\_ID (Primary Key)
  + SHOW\_TIME
  + SHOW\_DATE
  + LANGUAGE
  + MOVIE\_ID (Foreign Key)
  + THEATRE\_ID (Foreign Key)

## Payment

* **Attributes:**
  + PAYMENT\_ID (Primary Key)
  + TICKET\_NO (Foreign Key)
  + AMOUNT
  + PAYMENT\_DATE
  + PAYMENT\_METHOD

## Relationships:

* + Enables financial traceability for each booking.

## Key Relationships and Multiplicities

* **Customer to Ticket**: One-to-Many
* **Ticket to Show**: Many-to-One
* **Movie to Show**: One-to-Many
* **Theatre to Show / Seat / Ticket**: One-to-Many
* **Seat to Ticket**: One-to-One (unique seat per ticket)
* **Ticket to Payment**: One-to-One
* **Show to Ticket**: One-to-Many

## Summary

This ER diagram for the Movie Ticket Booking System presents a structured, normalized view of how users, movies, theatres, seats, and transactions are interconnected. With clearly defined entity relationships such as BOOKS, SHOWS, and RESERVES, the system ensures reliable seat allocation, secure payments, and efficient schedule management. This data-driven architecture enhances the user experience while offering theatre administrators complete control over operations, making it a scalable and robust solution for modern cinema management

**CHAPTER 2**

**1.2 RELATION – SCHEMA**

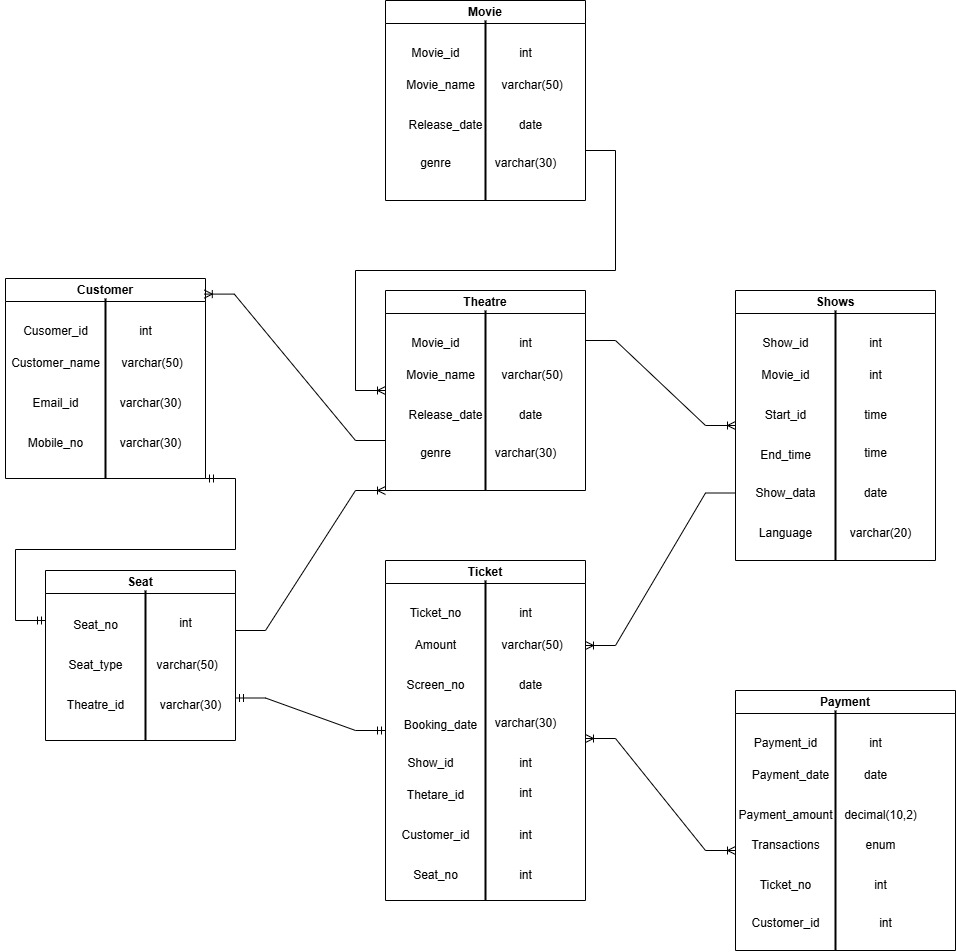


Fig:1.1 Relational Schema

2.1 This chapter includes:

* + - * Database Queries & Outputs: Execution of SQL queries with results.

**1.Theatre Table :**

CREATE TABLE Theatre (

TheatreID INT AUTO\_INCREMENT PRIMARY KEY,

TheatreName VARCHAR(100) NOT NULL,

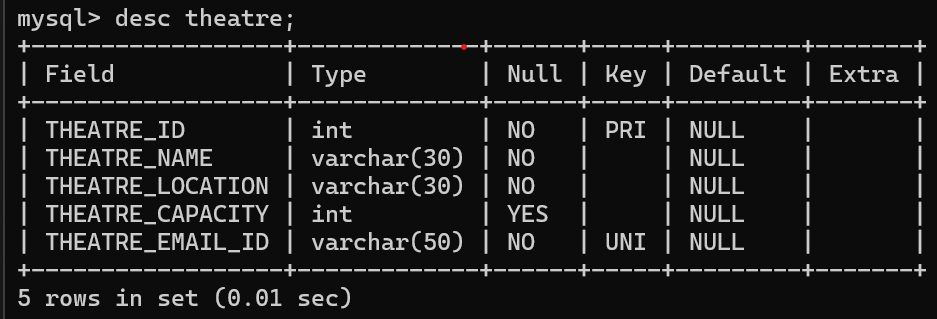
TheatreLocation VARCHAR(100) NOT NULL,

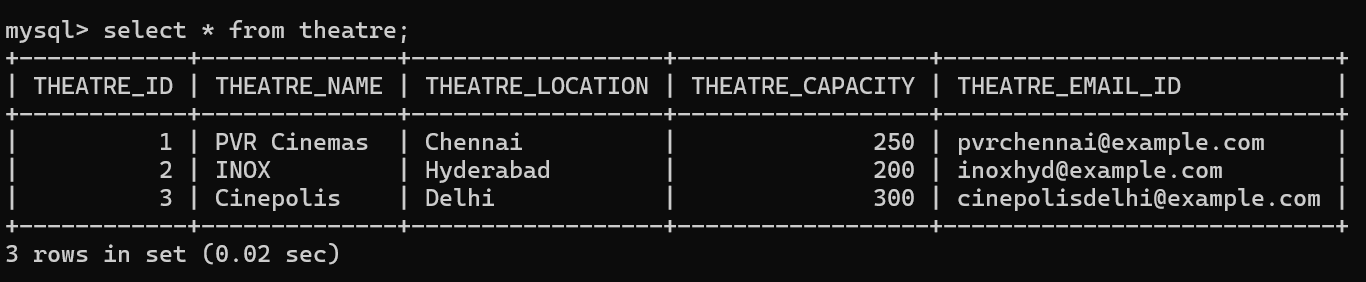
TheatreCapacity INT,

TheatreEmailID VARCHAR(100) NOT NULL UNIQUE

);

**Output:**

****

****

**2.SEAT:**

CREATE TABLE Seat (

SeatNo INT AUTO\_INCREMENT PRIMARY KEY,

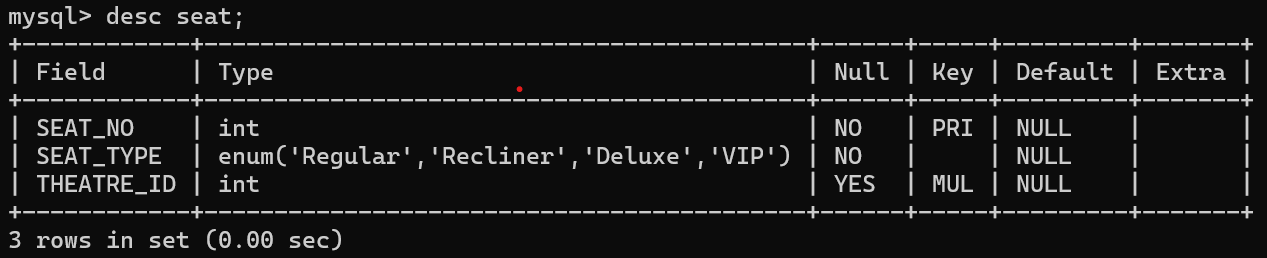
SeatType ENUM('Regular', 'Recliner', 'Deluxe', 'VIP') NOT NULL,

TheatreID INT NOT NULL,

FOREIGN KEY (TheatreID) REFERENCES Theatre(TheatreID) ON DELETE CASCADE

);

Output:

****

###### 

**3.Customer Table Schema:**

CREATE TABLE CUSTOMER (

customer\_id INT PRIMARY KEY AUTO\_INCREMENT,

first\_name VARCHAR(100) NOT NULL,

last\_name VARCHAR(100) NOT NULL,

email VARCHAR(100) UNIQUE NOT NULL,

phone\_number VARCHAR(15),

date\_of\_birth DATE,

registration\_date TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

status ENUM('active', 'inactive', 'suspended') DEFAULT 'active'

);

Output:

###### 

###### 

4.Movie Table Schema:

CREATE TABLE MOVIE (

movie\_id INT PRIMARY KEY AUTO\_INCREMENT,

title VARCHAR(255) NOT NULL,

genre VARCHAR(50),

director VARCHAR(100),

release\_date DATE,

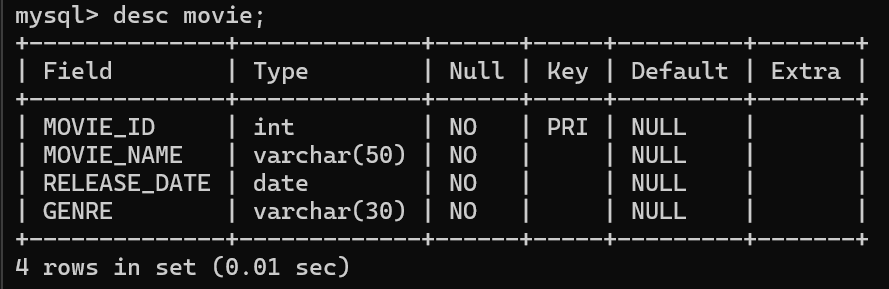
duration INT,

rating DECIMAL(3, 1),

language VARCHAR(50),

description TEXT

);

Output:  
  


###### 

**5.Shows Table Schema:**

CREATE TABLE SHOWS (

show\_id INT PRIMARY KEY AUTO\_INCREMENT,

movie\_id INT,

show\_time TIMESTAMP,

available\_seats INT,

theater\_id INT,

FOREIGN KEY (movie\_id) REFERENCES MOVIE (movie\_id)

);

Output:

###### 

6.Ticket Table Schema:

CREATE TABLE TICKET (

ticket\_id INT PRIMARY KEY AUTO\_INCREMENT,

customer\_id INT,

show\_id INT,

seat\_number VARCHAR (10),

booking\_time TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

price DECIMAL (10, 2),

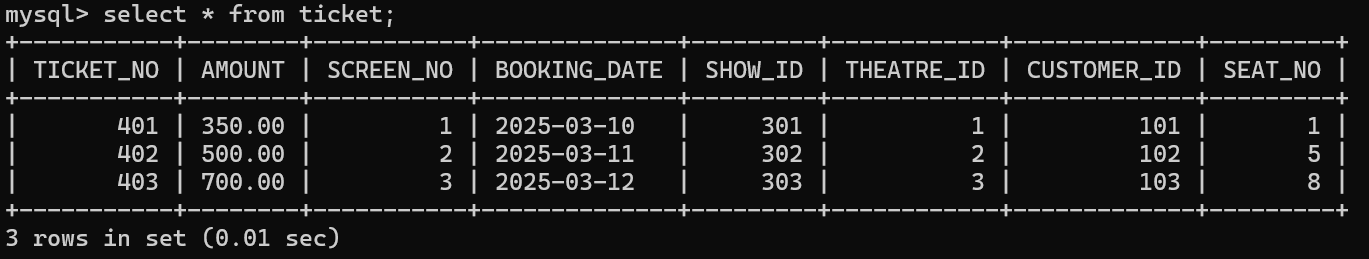
status ENUM ('booked', 'canceled', 'pending') DEFAULT 'booked',

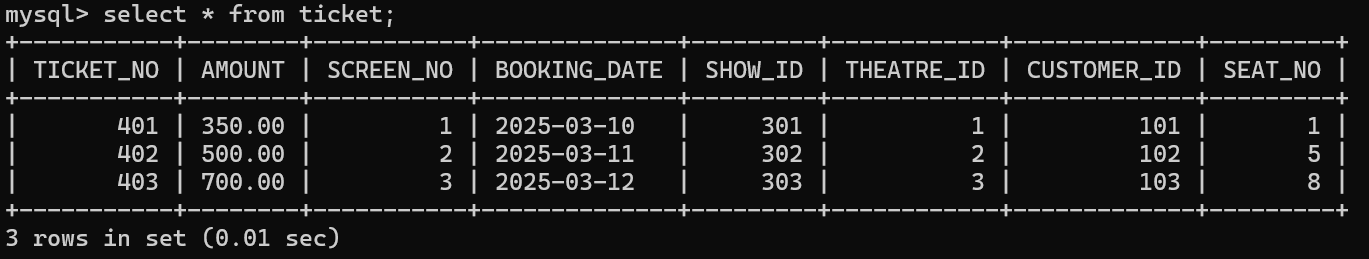
FOREIGN KEY (customer\_id) REFERENCES CUSTOMER (customer\_id),

FOREIGN KEY (show\_id) REFERENCES SHOWS (show\_id)

);

Output:





**7.Payment Table Schema:**

CREATE TABLE PAYMENT (

payment\_id INT PRIMARY KEY AUTO\_INCREMENT,

ticket\_id INT,

payment\_date TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

amount DECIMAL (10, 2),

payment\_method ENUM ('credit\_card', 'debit\_card', 'paypal', 'cash') NOT NULL,

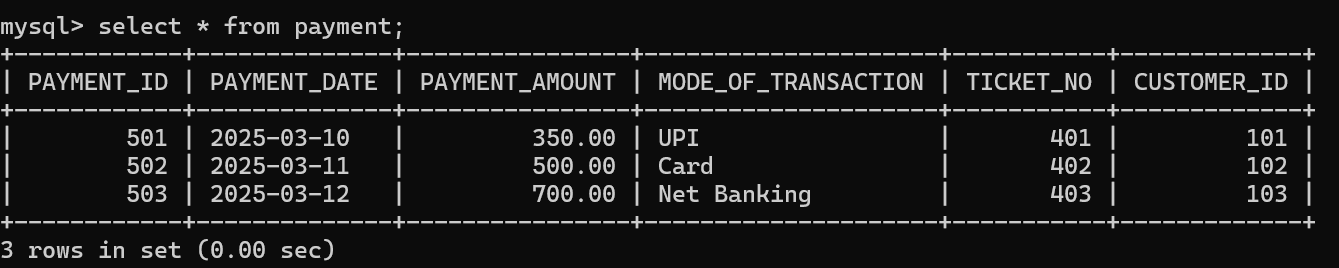
payment\_status ENUM ('completed', 'failed', 'pending') DEFAULT 'completed',

FOREIGN KEY (ticket\_id) REFERENCES TICKET (ticket\_id)

);

**Output:**

****

****

**2.2CRUD (Create, Read, Update, Delete) Operations**

The system supports CRUD operations for managing movie bookings.

* **Create: creating the tables**

1. CREATE TABLE Table\_name (name varchar (30),age int);  
   Theatre Table Schema

(The following displays the schema of the Theatre table, which contains details

about different theatres, including their name, location, capacity, and email ID)

CREATE TABLE Theatre (

THEATRE\_ID INT PRIMARY KEY,

THEATRE\_NAME VARCHAR(30) NOT NULL,

THEATRE\_LOCATION VARCHAR(30) NOT NULL,

THEATRE\_CAPACITY INT,

THEATRE\_EMAIL\_ID VARCHAR(50) NOT NULL UNIQUE

);

1. **Seat Table Schema**

(The following shows the schema of the Seat table, which stores seat details such as seat number, type, and the theatre it belongs to.)

CREATE TABLE Seat (

SEAT\_NO INT PRIMARY KEY,

SEAT\_TYPE ENUM('Regular', 'Recliner', 'Deluxe', 'VIP') NOT NULL,

THEATRE\_ID INT,

FOREIGN KEY (THEATRE\_ID) REFERENCES Theatre(THEATRE\_ID)

);

**3.Customer Table Schema**

(The following displays the schema of the Customer table, which contains customer information, including name, email ID, and mobile number.)

CREATE TABLE Customer (

CUSTOMER\_ID INT PRIMARY KEY,

CUSTOMER\_NAME VARCHAR(50) NOT NULL,

EMAIL\_ID VARCHAR(50) NOT NULL UNIQUE,

MOBILE\_NO VARCHAR(10) NOT NULL

);

**4.Movie Table Schema**

**(The following shows the schema of the Movie table, which stores movie details such as movie ID, name, release date, and genre).**CREATE TABLE Movie (

MOVIE\_ID INT PRIMARY KEY,

MOVIE\_NAME VARCHAR(50) NOT NULL,

RELEASE\_DATE DATE NOT NULL,

GENRE VARCHAR(30) NOT NULL

);

**5.Shows Table Schema**

(The following screenshot displays the schema of the Shows table, which contains details about movie shows, including show timings, date, language, and associated movie.)

CREATE TABLE Shows (

SHOW\_ID INT PRIMARY KEY,

SHOW\_TIME TIME NOT NULL,

SHOW\_DATE DATE NOT NULL,

MOVIE\_ID INT,

THEATRE\_ID INT,

FOREIGN KEY (MOVIE\_ID) REFERENCES Movie(MOVIE\_ID),

FOREIGN KEY (THEATRE\_ID) REFERENCES Theatre(THEATRE\_ID));

**6.Ticket Table Schema**

(The following screenshot shows the schema of the Ticket table, which contains booking details such as ticket number, amount, screen number, booking date, show ID, and theatre ID.)

CREATE TABLE Ticket (

TICKET\_NO INT PRIMARY KEY,

AMOUNT DECIMAL(10,2) NOT NULL,

SCREEN\_NO INT NOT NULL,

BOOKING\_DATE DATE NOT NULL,

SHOW\_ID INT,

THEATRE\_ID INT,

FOREIGN KEY (SHOW\_ID) REFERENCES Shows(SHOW\_ID),

FOREIGN KEY (THEATRE\_ID) REFERENCES Theatre(THEATRE\_ID)

);

**7.Payment Table Schema**

The Payment table stores transaction details related to ticket bookings. It maintains records of payments made by customers for their booked tickets.  
  
CREATE TABLE payment (

payment\_id INT PRIMARY KEY,

ticket\_no INT,

amount DECIMAL(10,2) NOT NULL,

payment\_date DATE NOT NULL,

payment\_method VARCHAR(50) NOT NULL,

FOREIGN KEY (ticket\_no) REFERENCES ticket(ticket\_no)

Insert : Inserting data in to tables:

insert into table\_name values(“raju”,20);

Read (SELECT): Retrieves data from tables.:

SELECT \* FROM MOVIE WHERE GENRE = 'Action';

Update (UPDATE): Modifies existing records.  
  
UPDATE SEAT SET SEAT\_TYPE = 'Deluxe' WHERE SEAT\_NO = 10;

Delete (DELETE): Removes records from tables.  
  
DELETE FROM TICKET WHERE TICKET\_NO = 105;

# Chapter 3

# Complex queries based on the concepts of constraints, sets, joins, views, Triggers and Cursors.

**3.1 Constraints:**

**1. PRIMARY KEY**

**2. FOREIGN KEY**

**3. NOT NULL**

**4. UNIQUE**

**5. CHECK**

🡪 creating primary key

CREATE TABLE CUSTOMER (

CUSTOMER\_ID INT PRIMARY KEY,

NAME VARCHAR(50),

EMAIL\_ID VARCHAR(50),

MOBILE\_NO VARCHAR(15)

);

**2.foregin key:**

**-->Taking foregin key reference from customer**

**🡪**CREATE TABLE TICKET (

TICKET\_NO INT PRIMARY KEY,

AMOUNT DECIMAL(10,2),

CUSTOMER\_ID INT,

FOREIGN KEY (CUSTOMER\_ID) REFERENCES CUSTOMER(CUSTOMER\_ID)

);

**3.Check constraints**

**🡪 Check payment\_amount should be greater than zero**

CREATE TABLE PAYMENT (

PAYMENT\_ID INT PRIMARY KEY,

PAYMENT\_AMOUNT DECIMAL(10,2),

MODE\_OF\_TRANSACTION VARCHAR(30),

CHECK (PAYMENT\_AMOUNT > 0)

);

**4.Not null**name VARCHAR(50) NOT NULL

**3.2Joins:**

**1.Inner joins**

Usage: for Returns only matching records in both tables.

SELECT C.CUSTOMER\_NAME, P.PAYMENT\_AMOUNT

FROM CUSTOMER C

INNER JOIN PAYMENT P ON C.CUSTOMER\_ID = P.CUSTOMER\_ID;

**Output:**

+----------------+----------------+

| customer\_\_name | PAYMENT\_AMOUNT |

+----------------+----------------+

| Rahul Sharma | 500.00 |

| Ayesha Khan | 700.00 |

| Kunal Mehta | 900.00 |

| Priya Verma | 1200.00 |

| Vikram Singh | 650.00 |

| Neha Patil | 800.00 |

| Rohan Das | 1000.00 |

| Arjun Reddy | 1200.00 |

+----------------+----------------+

8 rows in set (0.00 sec)

**2. LEFT JOIN:**

Returns all rows from the left table (CUSTOMER), and matching rows from the right (PAYMENT).

SELECT C.CUSTOMER\_NAME, P.PAYMENT\_AMOUNT

FROM CUSTOMER C

LEFT JOIN PAYMENT P ON C.CUSTOMER\_ID = P.CUSTOMER\_ID;

+----------------+----------------+

| customer\_\_name | PAYMENT\_AMOUNT |

+----------------+----------------+

| Rahul Sharma | 500.00 |

| Ayesha Khan | 700.00 |

| Kunal Mehta | 900.00 |

| Priya Verma | 1200.00 |

| Vikram Singh | 650.00 |

| Neha Patil | 800.00 |

| Rohan Das | 1000.00 |

| Arjun Reddy | 1200.00 |

+----------------+----------------+

8 rows in set (0.00 sec)

**3. RIGHT JOIN**

Returns all rows from the right table (PAYMENT), and matching rows from the left (CUSTOMER).

+----------------+----------------+

| customer\_\_name | PAYMENT\_AMOUNT |

+----------------+----------------+

| Rahul Sharma | 500.00 |

| Ayesha Khan | 700.00 |

| Kunal Mehta | 900.00 |

| Priya Verma | 1200.00 |

| Vikram Singh | 650.00 |

| Neha Patil | 800.00 |

| Rohan Das | 1000.00 |

| Arjun Reddy | 1200.00 |

+----------------+----------------+

8 rows in set (0.00 sec)

**3.3Sets:** In SQL, sets refer to collections of rows retrieved from one or more tables.  
Set operations allow you to combine, compare, or manipulate the results of two or more SELECT queries.

**1.Union**

🡪Get a list of all customer IDs from both TICKET and PAYMENT tables

SELECT CUSTOMER\_ID FROM TICKET

UNION

SELECT CUSTOMER\_ID FROM PAYMENT;

+-------------+

| CUSTOMER\_ID |

+-------------+

| 201 |

| 202 |

| 203 |

| 204 |

| 205 |

| 206 |

| 207 |

| 208 |

+-------------+

8 rows in set (0.06 sec)

**2.Union all**

🡪Same as above, but this time include duplicates

SELECT CUSTOMER\_ID FROM TICKET

UNION ALL

SELECT CUSTOMER\_ID FROM PAYMENT;

+-------------+

| CUSTOMER\_ID |

+-------------+

| 201 |

| 202 |

| 203 |

| 204 |

| 205 |

| 206 |

| 207 |

| 208 |

| 201 |

| 202 |

| 203 |

| 204 |

| 205 |

| 206 |

| 207 |

| 208 |

+-------------+

**3.Intersect**

🡪 Get customer IDs who booked a ticket and also made a payment

SELECT CUSTOMER\_ID FROM TICKET

WHERE CUSTOMER\_ID IN (

SELECT CUSTOMER\_ID FROM PAYMENT

);

+-------------+

| CUSTOMER\_ID |

+-------------+

| 201 |

| 202 |

| 203 |

| 204 |

| 205 |

| 206 |

| 207 |

| 208 |

+-------------+

8 rows in set (0.05 sec)

**3.4 Views:** A view in SQL is a virtual table based on the result of a SELECT query.  
It does not store data physically but displays data from one or more real tables.

**1.Creating the views**

CREATE VIEW CustomerPaymentSummary AS

SELECT C.customer\_\_name, P.PAYMENT\_AMOUNT, P.MODE\_OF\_TRANSACTION

FROM CUSTOMER C

JOIN PAYMENT P ON C.CUSTOMER\_ID = P.CUSTOMER\_ID;

**2.Viewing the view**

**SELECT \* FROM CustomerPaymentSummary;**

**+----------------+----------------+---------------------+**

| customer\_\_name | PAYMENT\_AMOUNT | MODE\_OF\_TRANSACTION |

+----------------+----------------+---------------------+

| Rahul Sharma | 500.00 | UPI |

| Ayesha Khan | 700.00 | Card |

| Kunal Mehta | 900.00 | Net Banking |

| Priya Verma | 1200.00 | Cash |

| Vikram Singh | 650.00 | UPI |

| Neha Patil | 800.00 | Card |

| Rohan Das | 1000.00 | Net Banking |

| Arjun Reddy | 1200.00 | Card |

+----------------+----------------+---------------------+

8 rows in set (0.05 sec)

**3.5 Triggers:** A trigger is a stored procedure in SQL that automatically executes (fires) in response to certain events on a table or view, such as INSERT, UPDATE, or DELETE.

**Step 1: create ticket\_log table**

CREATE TABLE TICKET\_LOG (

LOG\_ID INT AUTO\_INCREMENT PRIMARY KEY,

TICKET\_NO INT,

CUSTOMER\_ID INT,

LOG\_TIME TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

);

**Step2: Creating the triggers**

An **AFTER INSERT** trigger is executed automatically after a new row is inserted into a table. It is used to perform actions such as logging, updating related tables, or enforcing business rules after the insert operation.

DELIMITER $$

CREATE TRIGGER log\_ticket\_booking

AFTER INSERT ON TICKET

FOR EACH ROW

BEGIN

INSERT INTO TICKET\_LOG (TICKET\_NO, CUSTOMER\_ID)

VALUES (NEW.TICKET\_NO, NEW.CUSTOMER\_ID);

END$$

**DELIMITER ;**

**Step3: Inserting the values in it**

INSERT INTO TICKET (TICKET\_NO, AMOUNT, SCREEN\_NO, BOOKING\_DATE, SHOW\_ID, THEATRE\_ID, CUSTOMER\_ID, SEAT\_NO)

VALUES

(603, 750.00, 4, '2024-04-14', 401, 101, 202, 2),

(604, 1100.00, 1, '2024-04-14', 402, 102, 203, 3),

(605, 950.00, 2, '2024-04-14', 403, 103, 204, 4),

(606, 650.00, 5, '2024-04-14', 404, 104, 205, 5),

(607, 700.00, 3, '2024-04-14', 405, 105, 206, 6),

(608, 1000.00, 6, '2024-04-14', 406, 106, 207, 7),

(609, 600.00, 1, '2024-04-14', 407, 107, 208, 8);

**Step 4: Viewing the triggers**

SELECT \* FROM TICKET\_LOG;

+--------+-----------+-------------+---------------------+

| LOG\_ID | TICKET\_NO | CUSTOMER\_ID | LOG\_TIME |

+--------+-----------+-------------+---------------------+

| 1 | 601 | 201 | 2025-04-19 14:36:11 |

| 2 | 603 | 202 | 2025-04-19 14:36:49 |

| 3 | 604 | 203 | 2025-04-19 14:36:49 |

| 4 | 605 | 204 | 2025-04-19 14:36:49 |

| 5 | 606 | 205 | 2025-04-19 14:36:49 |

| 6 | 607 | 206 | 2025-04-19 14:36:49 |

| 7 | 608 | 207 | 2025-04-19 14:36:49 |

| 8 | 609 | 208 | 2025-04-19 14:36:49 |

+--------+-----------+-------------+---------------------+

8 rows in set (0.01 sec)

A **BEFORE INSERT** trigger is executed automatically before a new row is inserted into a table. It is commonly used for validation, modifying the inserted data, or checking conditions before allowing the insert.

DELIMITER $$

CREATE TRIGGER validate\_payment\_amount

BEFORE INSERT ON Payment

FOR EACH ROW

BEGIN

IF NEW.Amount <= 0 THEN

SIGNAL SQLSTATE '45000'

SET MESSAGE\_TEXT = 'Payment amount must be positive';

END IF;

END$$

DELIMITER ;

CREATE TABLE CancelledTickets (

CancelID INT AUTO\_INCREMENT PRIMARY KEY,

TicketNo INT,

CancelledOn DATETIME DEFAULT CURRENT\_TIMESTAMP

);

An **AFTER-DELETE** trigger is executed automatically **after** a row is deleted from a table. It is commonly used for tasks like logging, updating related tables, or enforcing referential integrity after a row is deleted.

DELIMITER $$

CREATE TRIGGER log\_ticket\_cancellation

AFTER DELETE ON Ticket

FOR EACH ROW

BEGIN

INSERT INTO CancelledTickets (TicketNo) VALUES (OLD.TicketNo);

END$$

DELIMITER ;

+----------+-----------+---------------------+

| CancelID | TicketNo | CancelledOn |

+----------+-----------+---------------------+

| 1 | 603 | 2025-05-05 15:00:00 |

+----------+-----------+---------------------+

CREATE TABLE SeatUpdates (

UpdateID INT AUTO\_INCREMENT PRIMARY KEY,

SeatNo INT,

OldType VARCHAR(50),

NewType VARCHAR(50),

UpdatedOn DATETIME DEFAULT CURRENT\_TIMESTAMP

);

An **AFTER-UPDATE** trigger is executed automatically after a row is updated in a table. It can be used for tasks such as logging changes, updating related tables, or enforcing business rules after the update.

DELIMITER $$

CREATE TRIGGER track\_seat\_changes

AFTER UPDATE ON Seat

FOR EACH ROW

BEGIN

IF OLD.SeatType != NEW.SeatType THEN

INSERT INTO SeatUpdates (SeatNo, OldType, NewType)

VALUES (OLD.SeatNo, OLD.SeatType, NEW.SeatType);

END IF;

END$$

DELIMITER ;

CREATE TABLE MovieLog (

LogID INT AUTO\_INCREMENT PRIMARY KEY,

MovieID INT,

MovieName VARCHAR(100),

LoggedAt DATETIME DEFAULT CURRENT\_TIMESTAMP

);

DELIMITER $$

CREATE TRIGGER log\_new\_movie

AFTER INSERT ON Movie

FOR EACH ROW

BEGIN

INSERT INTO MovieLog (MovieID, MovieName)

VALUES (NEW.MovieID, NEW.MovieName);

END$$

DELIMITER ;

DELIMITER $$

CREATE TRIGGER prevent\_customer\_delete

BEFORE DELETE ON Customer

FOR EACH ROW

BEGIN

IF EXISTS (

SELECT 1 FROM Ticket WHERE CustomerID = OLD.CustomerID

) THEN

**SIGNAL SQLSTATE '45000'**

**SET MESSAGE\_TEXT = 'Cannot delete customer with active bookings';**

**END IF;**

**END$$**

**DELIMITER ;**

**Cursors:** A cursor is a database object used to retrieve, process, and manipulate data row-by-row from a result set**.**

**Step1:Creating an customer\_log**

CREATE TABLE CUSTOMER\_LOG (

LOG\_ID INT AUTO\_INCREMENT PRIMARY KEY,

CUSTOMER\_NAME VARCHAR(50),

EMAIL\_ID VARCHAR(50),

LOG\_TIME TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

);

**Step2: create an cursor procedure**

DELIMITER $$

CREATE PROCEDURE LogCustomerDetails()

BEGIN

DECLARE done INT DEFAULT FALSE;

DECLARE cust\_name VARCHAR(50);

DECLARE cust\_email VARCHAR(50);

DECLARE cur CURSOR FOR

SELECT customer\_\_name, EMAIL\_ID FROM CUSTOMER;

DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;

DELETE FROM CUSTOMER\_LOG;

OPEN cur;

read\_loop: LOOP

FETCH cur INTO cust\_name, cust\_email;

IF done THEN

LEAVE read\_loop;

END IF;

INSERT INTO CUSTOMER\_LOG (CUSTOMER\_NAME, EMAIL\_ID)

VALUES (cust\_name, cust\_email);

END LOOP;

CLOSE cur;

END$$

DELIMITER ;

**Step3: Run the procedure**

**CALL LogCustomerDetails();**

Step 4: check the output

SELECT \* FROM CUSTOMER\_LOG;

+--------+---------------+------------------+---------------------+

| LOG\_ID | CUSTOMER\_NAME | EMAIL\_ID | LOG\_TIME |

+--------+---------------+------------------+---------------------+

| 1 | Rahul Sharma | rahul@gmail.com | 2025-04-19 14:48:09 |

| 2 | Ayesha Khan | ayesha@gmail.com | 2025-04-19 14:48:09 |

| 3 | Kunal Mehta | kunal@gmail.com | 2025-04-19 14:48:09 |

| 4 | Priya Verma | priya@gmail.com | 2025-04-19 14:48:09 |

| 5 | Vikram Singh | vikram@gmail.com | 2025-04-19 14:48:09 |

| 6 | Neha Patil | neha@gmail.com | 2025-04-19 14:48:09 |

| 7 | Rohan Das | rohan@gmail.com | 2025-04-19 14:48:09 |

| 8 | Arjun Reddy | arjun@gmail.com | 2025-04-19 14:48:09 |

+--------+---------------+------------------+---------------------+

8 rows in set (0.00 sec)

# Chapter 4

# Analyzing the pitfalls, identifying the dependencies, and applying normalizations

Normalization is the process of organizing data to eliminate redundancy and improve data integrity. It involves decomposing large, complex tables into smaller, simpler ones while maintaining relationships between them using keys.

## Normalization Stages in Your Project

Let's analyze your current schema and explain which **normal forms (1NF, 2NF, 3NF)** it satisfies.

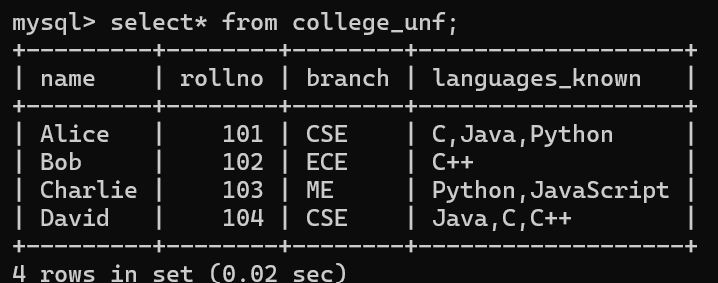
This Chapter includes Normal forms where it is used only for demonstration purpose but not

## 1NF (First Normal Form) Rule:

* Atomic values (no multi-valued or nested fields)
* Each column must contain only a single value
* Each record must be unique

## Your Tables:

* User,Donor,Patient,Transaction, etcare in **1NF** All fields contain atomic values (e.g.,Name,City,OrganRequired) Primary keys like UserID, TransactionID, PatientID ensure uniqueness



**Fig : 4.1**

## 

**Fig : 4.2**

## 2NF (Second Normal Form) Rule:

* Must be in 1NF
* No partial dependency (i.e., no non-key attribute depends only on part of a composite primary key)

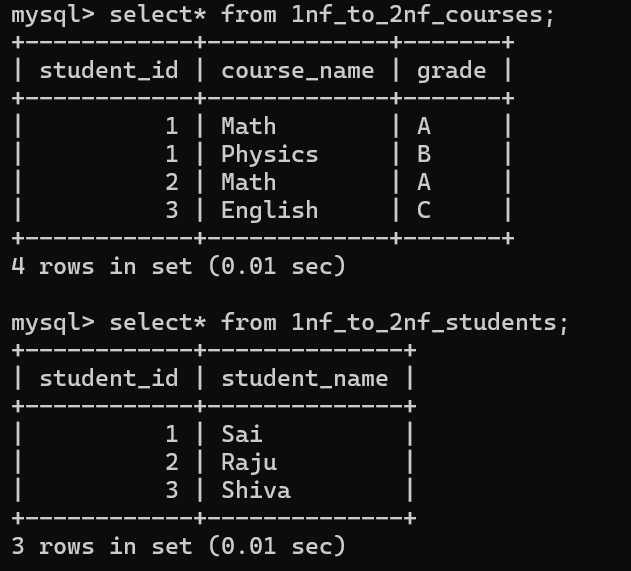
## Your Schema:

* Most tables use **single-column primary keys**, so **partial dependency doesn't exist** Example: In Donor, all non-key fields depend on the full primary key (DonorID) Transaction, Doctor, etc., follow this

**Note**: The use of surrogate keys (like DonorID, DoctorID) makes 2NF easier to achieve.

## 

## Fig : 4.3

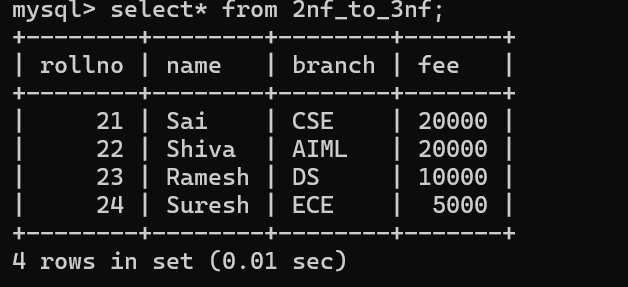


**Fig : 4.4**

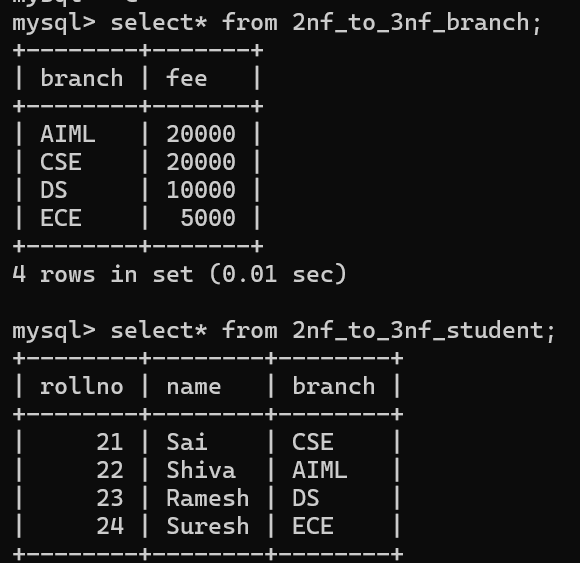
## 

## 3NF (Third Normal Form) Rule:

* Must be in 2NF
* No transitive dependencies (non-key attributes shouldn't depend on other non-key attributes)



**Fig : 4.5**

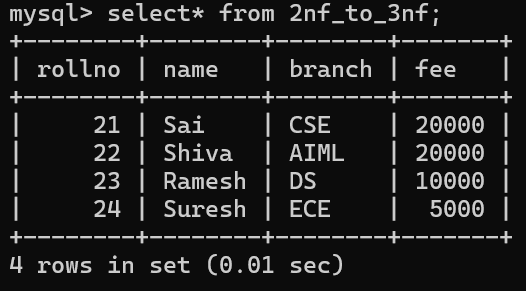
****

**Fig : 4.6**

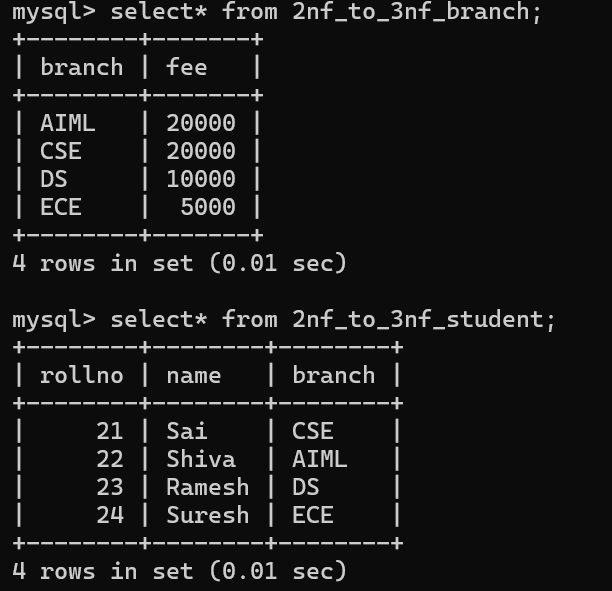
**4NF (Fourth Normal Form)**

* Remove **multivalued dependencies** (MVDs).

If a user can have **multiple medical histories and multiple insurances independently:**



**Fig : 4.5**



**Fig : 4.6**

**5NF (Fifth Normal Form)**

**Goal:**

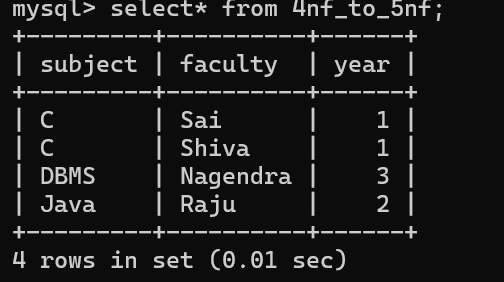
* Eliminate **join dependencies** that aren't implied by candidate keys.

Imagine we decompose even further:

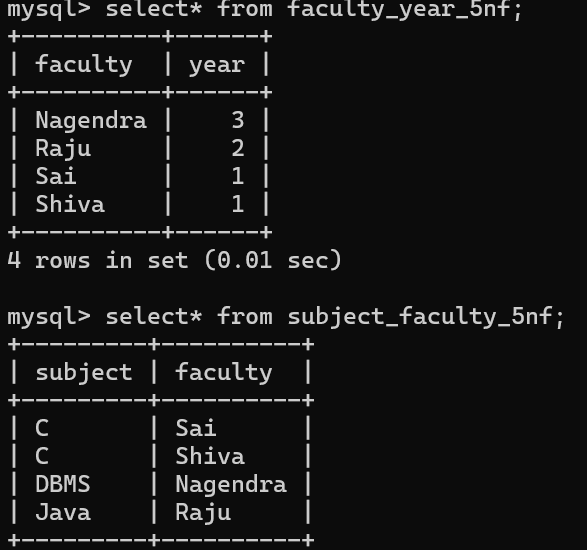
* One table for USER\_ID + MEDICAL\_HISTORY
* One for USER\_ID + MEDICAL\_INSURANCE
* One for USER\_ID + some\_third\_attribute

Only in rare complex scenarios does this break and require 5NF normalization. In your case,

**No such join dependencies**, so 5NF is already achieved with 4NF structure



**Fig : 4.7**

****

**Fig : 4.8**

# Chapter 5

# Implementation of concurrency control and recovery Mechanisms

Implementing concurrency control and recovery mechanisms in a database system like your Movie Ticket Booking System is essential to ensure data integrity, consistency, and fault tolerance during concurrent operations or system failures.

## Concurrency Control

Concurrency control ensures that multiple users can access and modify the database

**simultaneously** without conflicts or data corruption.

## Mechanisms:

* 1. **Transactions**

Wrap critical operations in transactions using START TRANSACTION, COMMIT,

and ROLLBACK.

START TRANSACTION;

UPDATE SHOW

SET available\_seats = available\_seats - 2

WHERE show\_id = 101;

INSERT INTO BOOKING (booking\_id, user\_id, show\_id, seats\_booked, booking\_time)

VALUES (301, 7, 101, 2, NOW())

COMMIT;

**If any operation fails, you can use ROLLBACK; to undo the entire transaction**.

* **ROLLBACK**

START TRANSACTION;

UPDATE SHOW SET available\_seats = available\_seats - 3 WHERE show\_id = 102;

ROLLBACK;

## Locking Mechanisms

Locks prevent multiple users from modifying the same data at the same time.

* + - **Implicit Locks**: Enabled automatically by InnoDB engine in MySQL.
    - **Explicit Locks** (rarely used directly):

**START TRANSACTION;**

SELECT available\_seats

FROM SHOW

WHERE show\_id = 101

FOR UPDATE;

UPDATE SHOW

SET available\_seats = available\_seats - 1

WHERE show\_id = 101;

COMMIT;

## Isolation Levels

Controls the visibility of changes made in one transaction to others. SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

Isolation Levels (from lowest to highest):

* + - **READ UNCOMMITTED**
    - **READ COMMITTED**
    - **REPEATABLE READ**
    - **SERIALIZABLE** (most strict, least concurrent)

SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;

START TRANSACTION;

SELECT \* FROM BOOKING WHERE show\_id = 101;

COMMIT;

* **READ COMMITTED**

SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

START TRANSACTION;

SELECT \* FROM BOOKING WHERE show\_id = 101;

COMMIT;

* **SERIALIZABLE**

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

START TRANSACTION;

SELECT \* FROM BOOKING WHERE show\_id = 101;

COMMIT;

## Recovery Mechanisms

Recovery mechanisms help restore the database to a consistent state after a crash or failure.

## Mechanisms:

* 1. **Transaction Logs (Write-Ahead Logging)**
     + Every change is written to a log before being applied.
     + If a crash occurs, logs are replayed to **redo committed transactions** and **undo uncommitted ones**.

This is handled **automatically** by DBMS engines like MySQL, PostgreSQL.

**CREATE TABLE TRANSACTION\_LOG (**

log\_id INT AUTO\_INCREMENT PRIMARY KEY,

user\_id INT,

action VARCHAR(255),

timestamp DATETIME DEFAULT CURRENT\_TIMESTAMP

);

INSERT INTO TRANSACTION\_LOG (user\_id, action)

VALUES (7, 'Booked 2 seats for show\_id 101');

## Backups and Restore

Regular backups protect against data loss. Example:

mysqldump -u root -p ramya2005 > ramya2005\_backup.sql # Restore

mysql -u root -p ramya2005 < ramya2005\_backup.sql

## Checkpointing

* + - Periodically saves the current state of the DB.
    - Speeds up recovery by reducing how much log data must be read after a crash.

CREATE TABLE SHOW\_CHECKPOINT AS SELECT \* FROM SHOW;

## Trigger for Recovery Log (Manual Approach)

Example trigger that log before deletion:

CREATE TABLE BOOKING\_BACKUP (  
 booking\_id INT,

user\_id INT,

show\_id INT,

seats\_booked INT,

deleted\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

);

CREATE TRIGGER before\_booking\_delete

BEFORE DELETE ON BOOKING

FOR EACH ROW

INSERT INTO BOOKING\_BACKUP (booking\_id, user\_id, show\_id, seats\_booked)

VALUES (OLD.booking\_id, OLD.user\_id, OLD.show\_id, OLD.seats\_booked);

**Chapter 6**

**Code for the Frontend**

from flask import Flask, request, redirect

import mysql.connector

app = Flask(\_\_name\_\_)

# Connect to MySQL

conn = mysql.connector.connect(

host="localhost",

user="root",

password="your\_password", # <--- change this

database="DBMS\_PROJECT\_MOVIE"

)

cursor = conn.cursor(dictionary=True)

@app.route('/')

def home():

cursor.execute("SELECT \* FROM Movie")

movies = cursor.fetchall()

movie\_list = "<h2>Available Movies</h2><ul>"

for m in movies:

movie\_list += f"<li>{m['MovieName']} - {m['Genre']} <a href='/shows/{m['MovieID']}'>View Shows</a></li>"

movie\_list += "</ul>"

return movie\_list

@app.route('/shows/<int:movie\_id>')

def shows(movie\_id):

cursor.execute("""

SELECT s.ShowID, s.ShowTime, s.ShowDate, s.Language, t.TheatreName

FROM Shows s

JOIN Theatre t ON s.TheatreID = t.TheatreID

WHERE s.MovieID = %s

""", (movie\_id,))

shows = cursor.fetchall()

html = f"<h2>Shows for Movie ID {movie\_id}</h2><ul>"

for s in shows:

html += f"<li>{s['ShowDate']} {s['ShowTime']} at {s['TheatreName']} ({s['Language']}) <a href='/book/{s['ShowID']}'>Book</a></li>"

html += "</ul>"

return html

@app.route('/book/<int:show\_id>', methods=["GET", "POST"])

def book(show\_id):

if request.method == "POST":

customer\_id = request.form["customer\_id"]

screen\_no = request.form["screen\_no"]

seat\_no = request.form["seat\_no"]

amount = request.form["amount"]

cursor.execute("SELECT TheatreID FROM Shows WHERE ShowID = %s", (show\_id,))

theatre\_id = cursor.fetchone()['TheatreID']

cursor.execute("""

INSERT INTO Ticket (ScreenNo, Amount, BookingDate, ShowID, TheatreID, SeatNo, CustomerID)

VALUES (%s, %s, CURDATE(), %s, %s, %s, %s)

""", (screen\_no, amount, show\_id, theatre\_id, seat\_no, customer\_id))

conn.commit()

return redirect("/bookings")

cursor.execute("SELECT SeatNo, SeatType FROM Seat")

seats = cursor.fetchall()

seat\_options = "".join([f"<option value='{s['SeatNo']}'>{s['SeatNo']} - {s['SeatType']}</option>" for s in seats])

return f"""

<h2>Book Ticket for Show ID {show\_id}</h2>

<form method='POST'>

Customer ID: <input name='customer\_id' required><br>

Screen No: <input name='screen\_no' required><br>

Seat No: <select name='seat\_no'>{seat\_options}</select><br>

Amount: <input name='amount' required><br>

<input type='submit' value='Book'>

</form>

"""

@app.route('/bookings')

def bookings():

cursor.execute("""

SELECT t.TicketNo, m.MovieName, s.ShowDate, s.ShowTime, th.TheatreName, t.Amount

FROM Ticket t

JOIN Shows s ON t.ShowID = s.ShowID

JOIN Movie m ON s.MovieID = m.MovieID

JOIN Theatre th ON t.TheatreID = th.TheatreID

""")

rows = cursor.fetchall()

table = "<h2>All Bookings</h2><table border='1'><tr><th>Ticket</th><th>Movie</th><th>Date</th><th>Time</th><th>Theatre</th><th>Amount</th></tr>"

for r in rows:

table += f"<tr><td>{r['TicketNo']}</td><td>{r['MovieName']}</td><td>{r['ShowDate']}</td><td>{r['ShowTime']}</td><td>{r['TheatreName']}</td><td>{r['Amount']}</td></tr>"

table += "</table>"

return table

if \_\_name\_\_ == '\_\_main\_\_':

app.run(debug=True)

**Chapter 7**

**Results and Discussion**

## Database Integration

The Movie Ticket Booking System integrates seamlessly with a MySQL database to ensure structured storage, quick retrieval, and integrity of data related to movies, theatres, customers, shows, seats, and payments. A well-designed relational schema incorporating primary and foreign key constraints, triggers, and views provides a robust and reliable backend infrastructure. APIs built using Flask, Django, or similar web frameworks enable real-time interaction between the user interface and the database, allowing users to browse showtimes, select seats, and make secure bookings dynamically and efficiently.

## Functionality Highlights

* **User Registration**: Individuals can register as donors or patients by submitting personal and medical information. Data is validated and stored in the User, Donor, and Patient tables.
* **Login and Authentication**: A secure login mechanism validates credentials for users such as doctors, donors, patients, and administrators. Role-based access control ensures secure data segregation.
* **Doctor & Organization Management**: Doctors and organizations are recorded in dedicated tables, along with their affiliations, departments, and contact information for efficient assignment and coordination.
* **Organ Request and Donation Tracking**: The system manages the end-to-end process of organ procurement by tracking donations and requests. Cursors and triggers ensure proper logging and validation before processing updates.
* **Transaction Handling**: Fee transactions and related details are managed securely. SQL constraints ensure valid status updates and bill processing.

## Discussion

The **Movie Ticket Booking System** effectively meets its functional and technical objectives by simplifying the ticket reservation process, ensuring secure data handling, and enabling smooth theatre operations. The database is normalized to avoid redundancy, with **foreign key constraints** maintaining referential integrity across entities.

Advanced SQL features such as **triggers** (e.g., for seat status updates) and **views** (e.g., upcoming show listings) improve performance and usability. The modular design of the system allows for future enhancements like integrating loyalty programs, analytics dashboards, or AI-based recommendation engines.

Overall, the system lays a scalable foundation for digitizing movie ticket operations and enriching the user experience through robust backend support and clean frontend interaction.

## Security Measures

* **Role-Based Access Control**: Doctors, patients, and administrators have role- specific permissions to ensure that sensitive data is protected.
* **Data Encryption**: Sensitive records (e.g., personal details and transaction data) are encrypted to ensure data privacy and confidentiality.
* **Injection Prevention**: All SQL inputs are sanitized using prepared statements to prevent SQL injection attacks.
* **Audit Logging with Triggers**: Important actions, such as deletions in the Patient table, are logged using triggers into backup tables for future recovery.

## Data Integrity

* **Foreign Key Constraints**: Enforced across tables such as Donor, Patient, and Transaction to maintain referential integrity.
* **Input Validation**: Backend checks prevent invalid data entries, particularly for dates, ENUMs (e.g., medical insurance), and mandatory fields.
* **ACID Transactions**: Key operations are enclosed in transactions to ensure

## User Interface Design

* **Accessibility**: Designed with screen reader support and color contrast for users with accessibility needs.
* **Clean Dashboard Navigation**: Separate views for admins, doctors, and users allow intuitive navigation of donation, request, and history data.
* **Responsive Design**: The frontend is responsive and compatible with mobile browsers, facilitating use in medical environments.

## Challenges and Limitations

* **Medical Record Privacy**: Managing and securing sensitive patient and donor data requires rigorous compliance with health privacy regulations (e.g., HIPAA or GDPR).
* **Data Matching Accuracy**: Matching donor organs with recipients based on medical parameters requires advanced logic, which could be enhanced with AI in future versions.
* **Scalability for National Use**: Scaling the system to a national registry level would require optimization of search algorithms and load handling.
* **Real-time Data Updates**: In a medical emergency, real-time updates are critical, which may require WebSockets or live APIs for continuous data sync.

## Future Enhancements

* **AI-based Donor-Recipient Matching**: Machine learning models can be introduced to suggest the best match based on compatibility and urgency.
* **Mobile App for Donors/Patients**: A mobile app could let users register, update medical details, and track requests/donations.
* **Blockchain for Secure Transactions**: Blockchain integration can improve the transparency and security of medical transactions and donor tracking.
* **Integration with National Health Databases**: This would facilitate cross- institution Organ Sharing And Coordination.

# OUTPUT SCREESHOTS:

1. **Page for GUI**

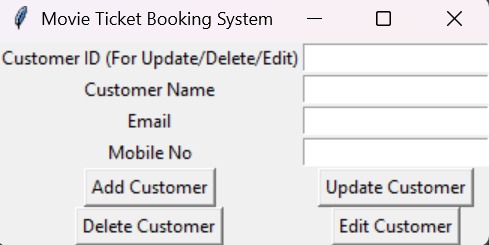


Fig :7.1

1. **ADDING OF CUSTOMER**

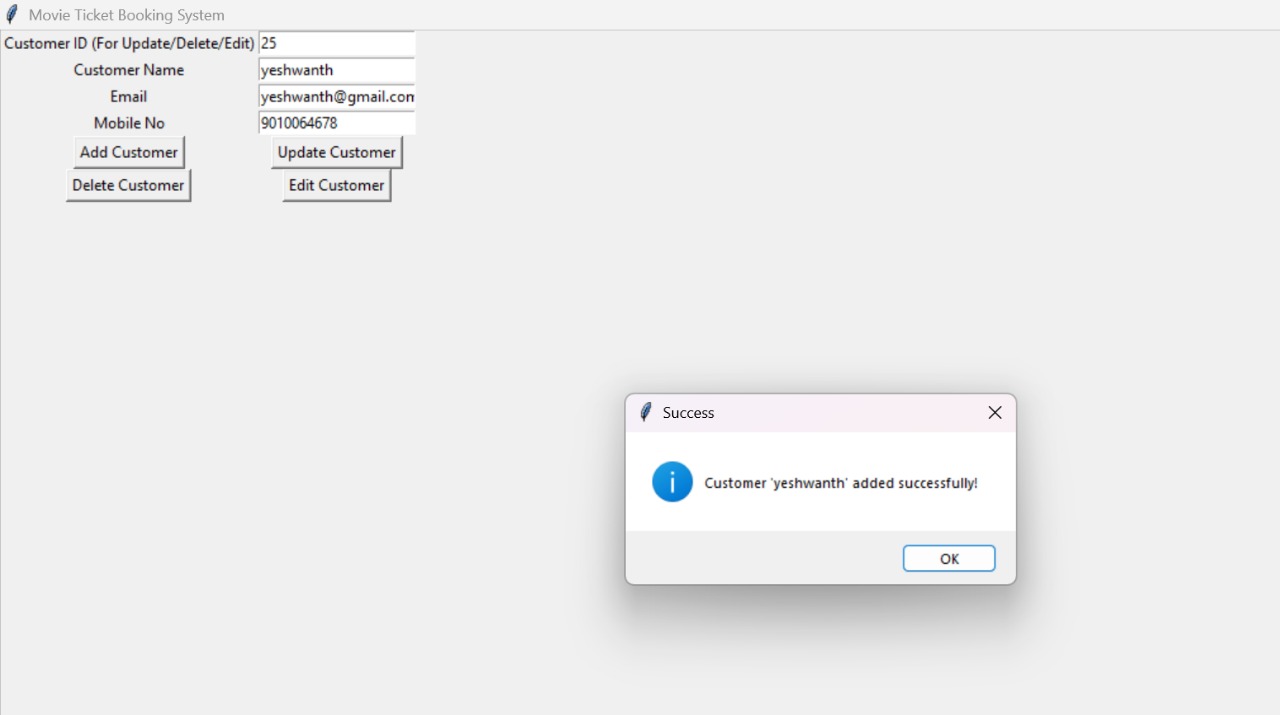


Fig:7.2

1. **IMPLIMENTED IN DATABASE**

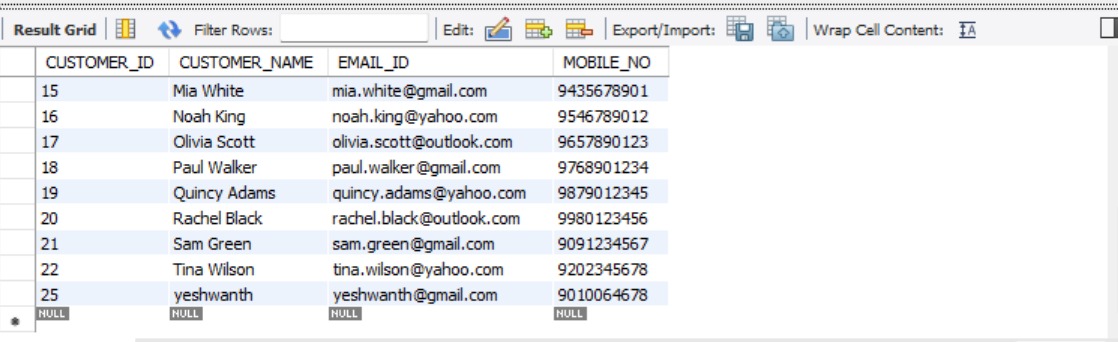


Fig:7.3

1. **UPDATE**

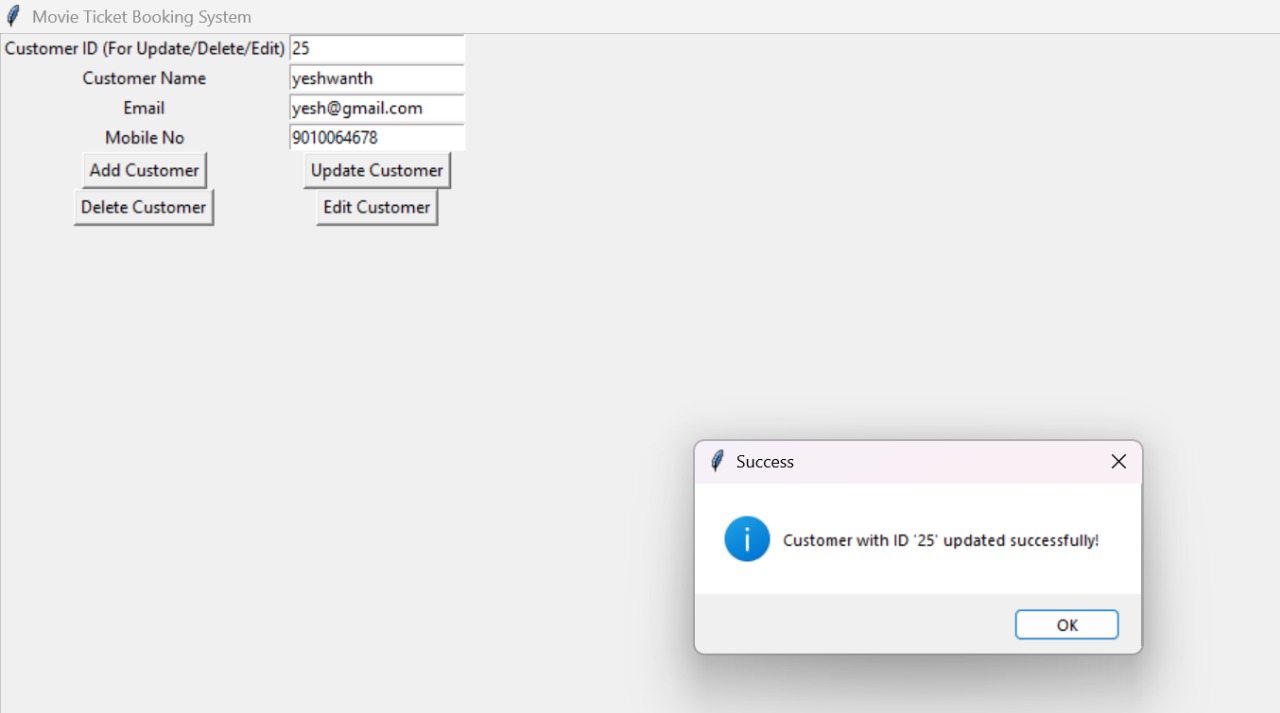
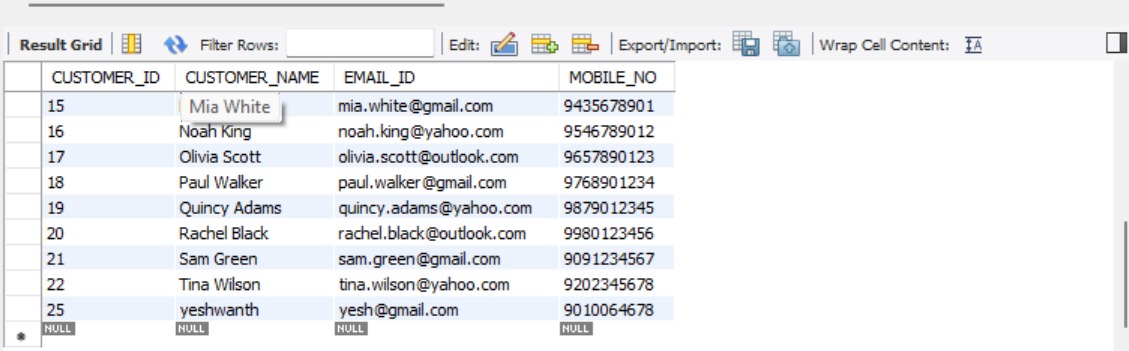


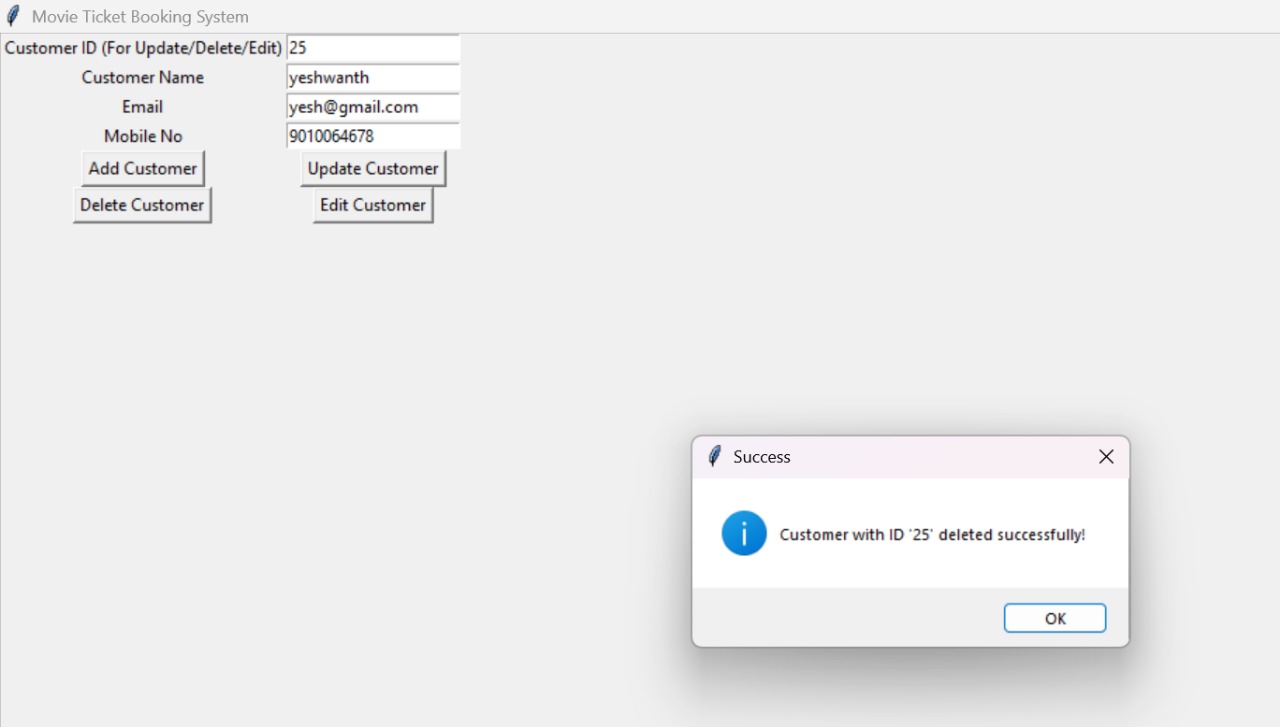
Fig:7.4

**5.IMPLIMENTING IN DATABASE**



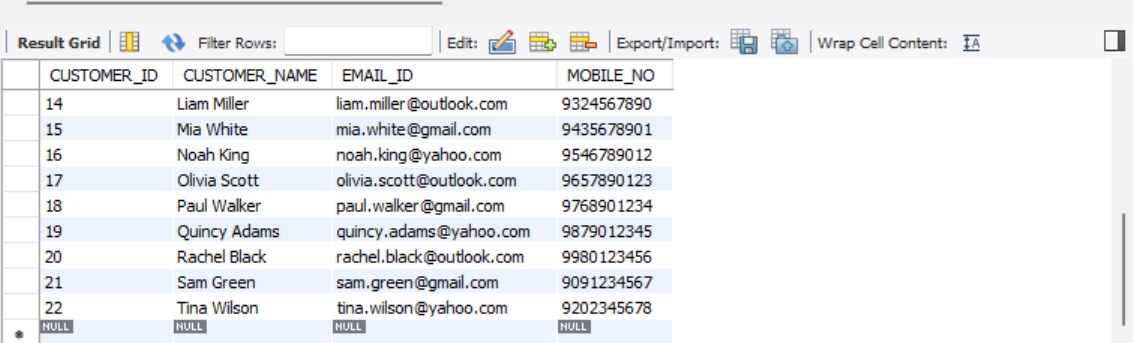
**Fig : 7.4**

**6.DELETING**



**Fig : 7.5**

**6.Implimenting in Database**



**Fig : 7.6**

**Conclusion**

The Movie Ticket Booking System effectively addresses the core requirements of managing movie screenings, seat reservations, and payment processing in a structured and user-friendly environment. By leveraging a relational MySQL database and well-designed tables such as Customer, Movie, Theatre, Show, Seat, Ticket, and Payment, the system ensures data integrity, operational efficiency, and secure user transactions.

The project demonstrates the potential of database-driven applications in streamlining real-world business operations. The use of primary and foreign keys, normalized schemas, and relational constraints ensures non-redundant, consistent data management. Integration of advanced SQL features such as joins, views, triggers, and stored procedures supports intelligent automation, fast query execution, and dynamic updates across the platform.

This system provides a scalable and modular foundation suitable for real-world deployment in multiplex chains, independent cinemas, or ticketing platforms. Future enhancements could include:

In summary, the Movie Ticket Booking System combines efficiency, usability, and scalability, paving the way for future innovations in digital cinema operations.

**7.4 REFERENCES**

• MySQL Documentation: https://dev.mysql.com/doc/

• Database Design Best Practices: https://www.databasejournal.com/

• https://medium.com/nerd-for-tech/system-design-of-a-Movie-booking-system-redbus-5703ce6e02db

• https://www.geeksforgeeks.org/how-to-design-database-for-logistics-and-Booking/

• https://avestia.com/CSEE2016\_Proceedings/files/paper/ICTE/107.pdf

• https://www.ijirem.org/DOC/4\_design-of-a-database-for-bus-fleet-management.pdf

• https://www.ey.com/en\_us/alliances/case-study-redefining-a-Movie-management-system

• Movie Corporation (India) Ltd