HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

DATABASE LAB - IT3290E

FINAL PROJECT REPORT: MOVIE TICKETING SYSTEM

Instructor: **Ph.D. Vu Tuyet Trinh**

Group: 2
Students: Dang Van Nhan - 20225990
Nguyen Lan Nhi - 20225991
Duong Phuong Thao - 20226001

Ha Noi, December 2024

Preface

In the era of Industry 4.0, the film industry has been developing rapidly, becoming an indispensable part of cultural and recreational life. The emergence of online streaming platforms, along with the constant innovation of traditional cinemas, has brought unique and captivating experiences to audiences. To support and enhance the user experience, the systematization of movie ticket sales has become crucial. An effective ticket management system not only optimizes cinema operations but also ensures convenience and speed for customers, thereby improving service quality.

Driven by these practical needs, our group (Group 2) chose the topic **"Movie Ticketing System"** for the **Database Lab** course as the Final Project. The goal of our project is to develop a system capable of supporting online ticket booking, managing customer and admin information, and improving operational efficiency.

We would like to express our heartfelt gratitude to **Ph.D. Vu Tuyet Trinh**, who dedicated her time and effort to guide, review, and support our team throughout the project. Her enthusiasm and invaluable feedback have been a great source of motivation for us to complete and refine our work.

However, due to the limited time and the team's lack of extensive experience, the project still contains several shortcomings. We sincerely look forward to receiving feedback and suggestions from our instructor and peers to further improve and develop this system in the future.

Contents

1 Contribution

This section highlights each group member's contributions to this project, providing an overview of the work they have contributed to. Member with **bold** text is the head of the group.

Member	Tasks
Dang Van Nhan - 20225990	• Design **Entity-Relationship Diagram** • Design **Relational Schema** • Choose, implement **indexes** for the database • Choose, implement **views** and **materialized views** for the database • Support the testing process • Complete the report
Nguyen Lan Nhi - 20225991	• Critique and suggest edits for **Entity-Relationship Diagram** • Implement **triggers** for the database • Prepare slide • Support the testing process • Complete the report
Duong Phuong Thao - 20226001	• Critique and suggest edits for **Relational Schema** • Implement **stored procedures** for the database • Implement **functions** for the database • Support the testing process • Complete the report

Table 1: Contribution of Group 2's members

2 Introduction

2.1 Description

The objective of this project is to design and implement a well-optimized database system for managing ticketing operations in a multi-branch cinema chain. The system is designed to efficiently manage cinemas, screening rooms, and movies, while enabling real-time showtime browsing and ticket booking. Additionally, it provides effective seat availability management and ensures accurate tracking of bookings. The database also supports generating detailed reports for analysis, facilitating data-driven decision-making and operational efficiency.

2.2 Business Requirements

The business requirements define the essential functionalities and features that the database system must support to ensure efficient management of ticketing operations in a multi-branch cinema chain. These requirements aim to address both user needs and administrative goals, focusing on functionality, performance, and usability.

- **Cinema and Showtime Management:** The system must support managing data for multiple cinema locations, allowing customized schedules for the same movie across different branches. Administrators should be able to update movie schedules, manage promotional campaigns, and issue vouchers seamlessly.

- **Movie Details and Real-Time Seat Selection:** The system should provide comprehensive movie information, including titles, genres, durations, and more. It must also include a real-time seating chart that allows users to view and select available seats efficiently.

- **Bookings and Payments:** The booking system should offer flexible workflows, enabling users to choose a cinema or movie first, select seats, and process payments through a secure third-party system. Seats must be held for a specific duration during the payment process. The third-party system will determine the pending payment status and return a result of either `"confirmed"` or `"cancelled"` to the database. Successful transactions should reward users with 5% of the total payment value in Loyalty Points.

- **Voucher Redemption:** The system must allow users to redeem vouchers using their Loyalty Points, provided they meet the required threshold. Redeemed vouchers should be tracked, applied to reduce ticket costs, and updated in real-time with statuses such as `"used"` or `"expired"`, depending on their usage or expiration date.

- **User Account Management:** The system should enable users to create accounts for managing their bookings, payment history, loyalty points, and personal information. This ensures a personalized and streamlined user experience.

- **Admin Reporting:** The system must provide detailed reports on critical metrics such as ticket sales, revenue, popular movies, and showtimes. These reports will help administrators optimize scheduling, enhance operational efficiency, and make data-driven decisions.

These requirements ensure that the database system fulfills the functional and operational needs of both users and administrators, delivering a seamless and efficient ticketing experience.

3 Database Design

This section focuses on the design of the database, starting with the Entity-Relationship Diagram (ERD) to model the core entities and their relationships in a movie ticketing system. The ERD is then systematically converted into a Relational Schema (RS), ensuring normalization and adherence to best practices. This process ensures that the database is well-structured, scalable, and efficient for managing data.

3.1 Entity-Relationship Diagram

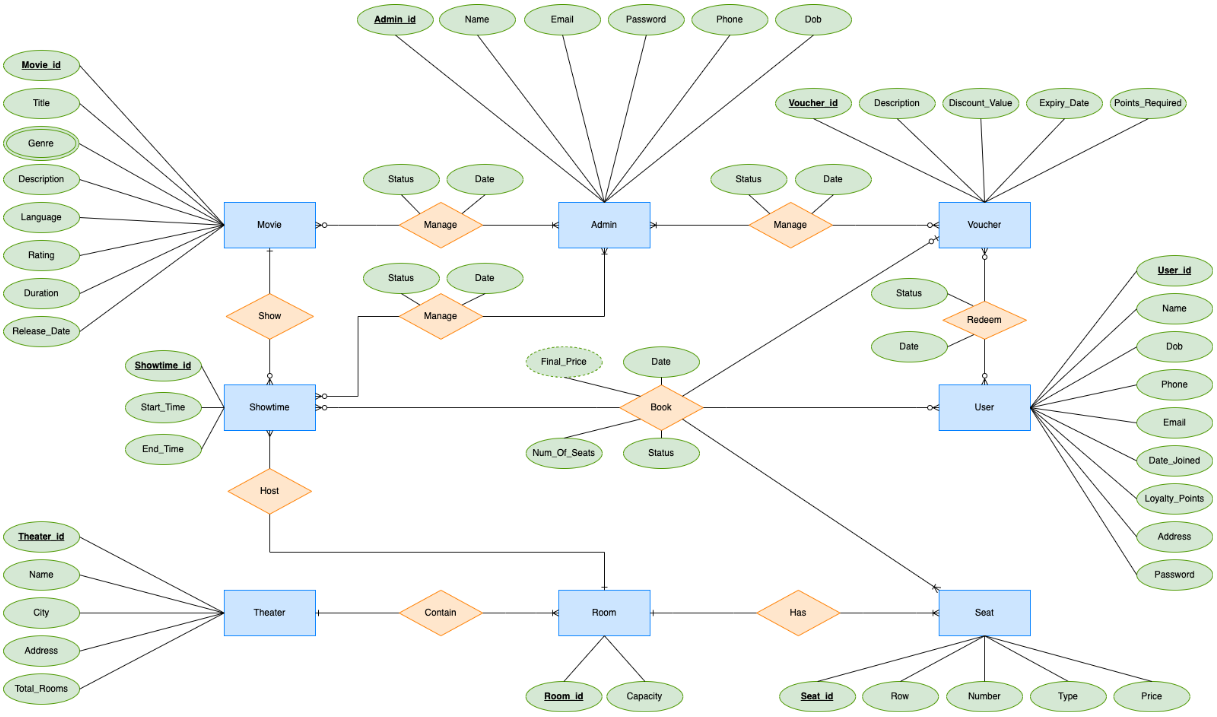

Figure 1: Entity-Relationship Diagram of Movie Ticketing System

This Entity-Relationship Diagram (ERD) is designed to model a movie ticket management system, comprising eight main entities: **Movie**, **Admin**, **User**, **Theater**, **Room**, **Seat**, **Showtime**, and **Voucher**. These entities are represented by **blue rectangles** in the diagram and are modeled as **strong entities** because each has a unique key attribute and does not fully depend on other entities.

The relationships between these entities, shown as **orange diamonds**, capture the interactions and dependencies, such as `Manage`, `Book`, or `Redeem`. Each relationship connects two or more entities and is annotated with attributes where necessary.

The attributes in **green ovals** provide details about the data stored in entities. For example, the **Movie** entity includes attributes such as `Title`, `Genre`, `Language`, and `Release_Date`. **Green ovals with bold and underlined text** represent **key attributes**, which uniquely identify each instance of an entity or a relationship. Notably, `Genre` is a **multivalued** attribute, represented by **green ovals with two concentric lines**, indicating multiple values can exist for a single entity instance. Additionally, the `Final_Price` attribute in the `Book` relationship is a **derived** attribute, calculated based on other attributes or relationships, represented by a **green oval with dashed lines**. These notations enhance the clarity and interpretation of the Entity-Relationship Diagram (ERD), facilitating accurate implementation.

The entities in the system interact with each other through several relationships. Notable relationships include **Show** (connecting **Movie** and **Showtime** to represent a specific screening of a movie), **Host** (linking **Showtime** with **Room** to indicate where the screening takes place), and **Contain** (connecting **Theater** with its rooms). Additionally, the **Has** relationship describes the seats in each room. Each relationship is carefully designed to reflect real-world operations, such as the **Manage** relationship allowing admins to manage movies, showtimes, and vouchers.

The **Book** relationship, which connects four entities: **User**, **Showtime**, **Seat**, and **Voucher**, is particularly noteworthy. This is a **quaternary** relationship that represents the complex ticket booking process, where users select a specific showtime, choose seats, and may apply a voucher for a discount. This relationship is essential for capturing all details of a booking transaction, such as the seat status (reserved or not), voucher status (used or valid), and which user made the booking. However, this relationship also adds complexity to the implementation, requiring careful design of database tables and efficient queries.

Overall, this ERD is detailed and thoroughly models key aspects of the movie ticket management system. Its strengths include clear information, rich relationships, and comprehensive modeling of the business requirements.

3.2 Relational Schema

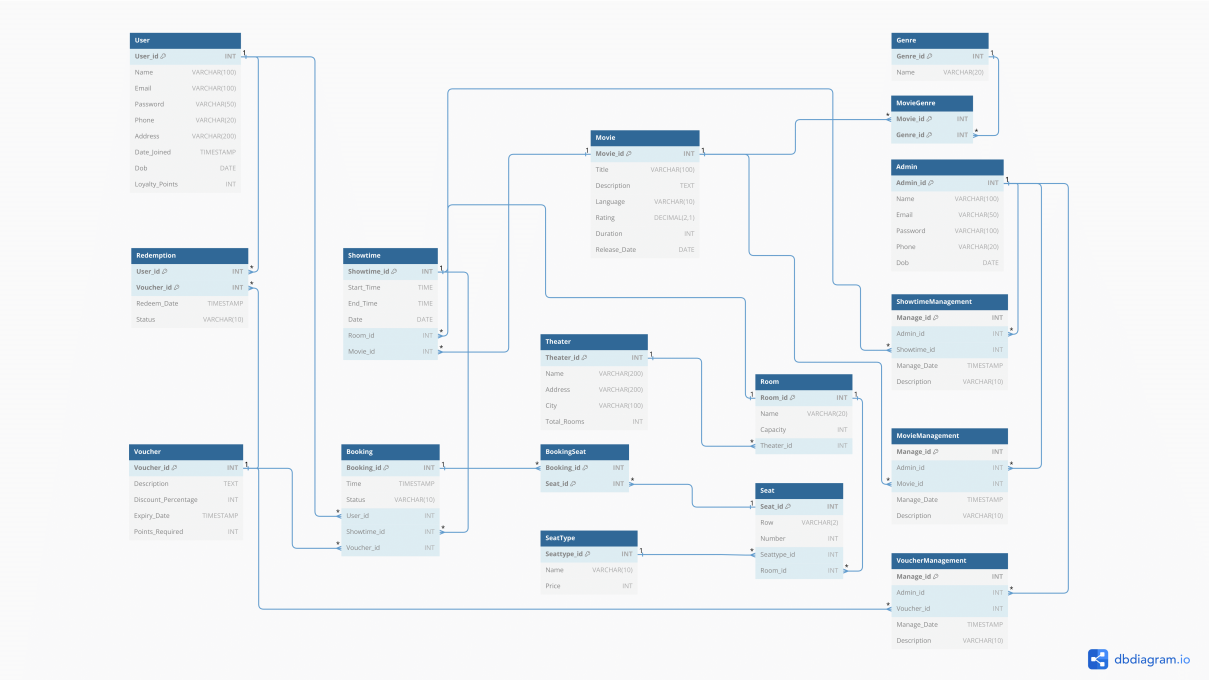

Figure 2: Relational Schema of Movie Ticketing System

Each entity in the ERD has been mapped to a corresponding table in the Relational Schema. For instance, entities such as `User`, `Movie`, `Theater`, `Room`, `Seat`, `Showtime`, and `Voucher` have been transformed into their respective tables. Attributes of these entities have been translated into columns with appropriate data types and constraints, such as `NOT NULL`, `CHECK IN`, `CHECK BETWEEN`, `PRIMARY KEY`, and `FOREIGN KEY`.

Many-to-many relationships in the ERD have been handled by introducing intermediate (associative) tables. For example, the `Redemption` table represents the many-to-many relationship between `User` and `Voucher`, while the `BookingSeat` table connects the relationship between

Booking and Seat.

The relational schema demonstrates a robust and systematic design that adheres to best practices in database normalization, particularly ensuring compliance with **Third Normal Form (3NF)**. The use of intermediate tables such as MovieGenre for the many-to-many relationship between Movie and Genre, and Redemption for the relationship between User and Voucher, reflects a strong effort to minimize data redundancy and simplify data retrieval. This separation of data ensures that each entity and relationship is represented uniquely, reducing the risk of inconsistencies or duplicate information.

From a data integrity perspective, the schema effectively employs foreign keys to enforce relationships between tables. For example, references like User(User_id) and Showtime(Showtime_id) are consistently applied to maintain logical connections across tables, thereby safeguarding the accuracy of relational data during updates or deletions. However, the dependency on foreign keys also increases the complexity of queries, which could pose challenges in large-scale operations.

In terms of scalability, the schema is designed to accommodate growth with the inclusion of management tables such as MovieManagement, ShowtimeManagement, and VoucherManagement. These tables provide an additional layer of administrative control, enabling detailed tracking of actions performed on critical data entities. While this adds flexibility to the system, it also introduces potential overhead in managing these additional tables, which may require further optimization as the dataset expands.

Overall, the relational schema is well-structured and demonstrates a balance between normalization, data integrity, and scalability. Below is the detailed information about the tables in the Relational Schema:

Field Name	**Data Type**	**Key**	**Constraints**	**Meaning**
User_id	INT	PRIMARY	NOT NULL	The unique identifier of the user
Name	VARCHAR(100)		NOT NULL	User's name
Email	VARCHAR(100)		UNIQUE, NOT NULL	User's email address
Password	VARCHAR(50)		NOT NULL	User's password for authentication
Phone	VARCHAR(20)			User's phone number
Address	VARCHAR(200)			User's physical address
Date_Joined	TIMESTAMP			The date and time the user joined
Dob	DATE			User's date of birth
Loyalty_Points	INT			Points earned by the user for redeeming vouchers

Table 2: Information of **User** table

Field Name	Data Type	Key	Constraints	Meaning
Voucher_id	INT	PRIMARY	NOT NULL	The unique identifier of the voucher
Description	TEXT			Details about the voucher
Discount_Percentage	INT		CHECK (Discount_Percentage BETWEEN 10 AND 100)	The discount percentage offered by the voucher
Expiry_Date	TIMESTAMP			The expiration date and time of the voucher
Points_Required	INT			The number of loyalty points required to redeem the voucher

Table 3: Information of **Voucher** table

Field Name	Data Type	Key	Constraints	Meaning
User_id	INT	COMPOSITE PRIMARY, FOREIGN	NOT NULL	The unique identifier of the user associated with the redemption. References to **User(User_id)**
Voucher_id	INT	COMPOSITE PRIMARY, FOREIGN	NOT NULL	The unique identifier of the voucher being redeemed. References to **Voucher(Voucher_id)**
Redeem_Date	TIMESTAMP		NOT NULL	The date and time of redemption
Status	VARCHAR(10)		CHECK (Status IN ('Available', 'Used', 'Expired'))	The status of the redemption

Table 4: Information of **Redemption** table

Field Name	Data Type	Key	Constraints	Meaning
Theater_id	INT	PRIMARY	NOT NULL	The unique identifier of the theater
Name	VARCHAR(200)		NOT NULL	The name of the theater

Address	VARCHAR(200)			The address of the theater
City	VARCHAR(100)			The city where the theater is located
Total_Rooms	INT			The total number of rooms in the theater

Table 5: Information of **Theater** table

Field Name	**Data Type**	**Key**	**Constraints**	**Meaning**
Room_id	INT	PRIMARY	NOT NULL	The unique identifier of the room
Name	VARCHAR(20)		NOT NULL	The name of the room
Capacity	INT			The seating capacity of the room
Theater_id	INT	FOREIGN		The identifier of the theater the room belongs to. References to **Theater(Theater_id)**

Table 6: Information of **Room** table

Field Name	**Data Type**	**Key**	**Constraints**	**Meaning**
Seattype_id	INT	PRIMARY	NOT NULL	The unique identifier of the seat type
Name	VARCHAR(10)			The name of the seat type
Price	INT			The price associated with the seat type

Table 7: Information of **SeatType** table

Field Name	**Data Type**	**Key**	**Constraints**	**Meaning**
Seat_id	INT	PRIMARY	NOT NULL	The unique identifier of the seat
Row	VARCHAR(2)		NOT NULL	The row where the seat is located
Number	INT		NOT NULL	The number of the seat within the row
Seattype_id	INT	FOREIGN		The identifier of the seat type. References to **SeatType(Seattype_id)**

Room_id	INT	FOREIGN		The identifier of the room the seat belongs to. References to **Room(Room_id)**

Table 8: Information of **Seat** table

Field Name	**Data Type**	**Key**	**Constraints**	**Meaning**
Movie_id	INT	PRIMARY	NOT NULL	The unique identifier of the movie
Title	VARCHAR(100)		NOT NULL	The title of the movie
Description	TEXT			A brief description of the movie
Language	VARCHAR(10)			The language in which the movie is available
Rating	DECIMAL(2,1)			The average user rating for the movie
Duration	INT			The duration of the movie in minutes
Release_Date	DATE			The release date of the movie

Table 9: Information of **Movie** table

Field Name	**Data Type**	**Key**	**Constraints**	**Meaning**
Genre_id	INT	PRIMARY	NOT NULL	The unique identifier of the genre
Name	VARCHAR(20)		NOT NULL	The name of the genre

Table 10: Information of **Genre** table

Field Name	**Data Type**	**Key**	**Constraints**	**Meaning**
Movie_id	INT	COMPOSITE PRIMARY, FOREIGN	NOT NULL	The unique identifier of the movie. References to **Movie(Movie_id)**
Genre_id	INT	COMPOSITE PRIMARY, FOREIGN	NOT NULL	The unique identifier of the genre. References to **Genre(Genre_id)**

Table 11: Information of **MovieGenre** table

Field Name	**Data Type**	**Key**	**Constraints**	**Meaning**

Showtime_id	INT	PRIMARY	NOT NULL	The unique identifier of the showtime
Start_Time	TIME		NOT NULL	The start time of the showtime
End_Time	TIME		NOT NULL	The end time of the showtime
Date	DATE		NOT NULL	The date of the showtime
Room_id	INT	FOREIGN		The unique identifier of the room. References to **Room(Room_id)**
Movie_id	INT	FOREIGN		The unique identifier of the movie. References to **Movie(Movie_id)**

Table 12: Information of **Showtime** table

Field Name	**Data Type**	**Key**	**Constraints**	**Meaning**
Booking_id	INT	PRIMARY	NOT NULL	The unique identifier of the booking
Time	TIMESTAMP		NOT NULL	The timestamp of the booking
Status	VARCHAR(10)		CHECK (Status IN (’Pending’, ’Confirmed’, ’Cancelled’))	The status of the booking
User_id	INT	FOREIGN	NOT NULL	The identifier of the user making the booking. References to **User(User_id)**
Showtime_id	INT	FOREIGN	NOT NULL	The identifier of the showtime being booked. References to **Showtime(Showtime_id)**
Voucher_id	INT	FOREIGN		The identifier of the voucher used in the booking. References to **Voucher(Voucher_id)**

Table 13: Information of **Booking** table

Field Name	Data Type	Key	Constraints	Meaning
Booking_id	INT	COMPOSITE PRIMARY, FOREIGN	NOT NULL	The unique identifier of the booking. References to **Booking(Booking_id)**
Seat_id	INT	COMPOSITE PRIMARY, FOREIGN	NOT NULL	The unique identifier of the seat. References to **Seat(Seat_id)**

Table 14: Information of **BookingSeat** table

Field Name	Data Type	Key	Constraints	Meaning
Admin_id	INT	PRIMARY	NOT NULL	The unique identifier of the admin
Name	VARCHAR(100)		NOT NULL	The name of the admin
Email	VARCHAR(50)		UNIQUE, NOT NULL	The email address of the admin
Password	VARCHAR(100)		NOT NULL	The password for the admin
Phone	VARCHAR(20)			The phone number of the admin
Dob	DATE			The date of birth of the admin

Table 15: Information of **Admin** table

Field Name	Data Type	Key	Constraints	Meaning
Manage_id	INT	PRIMARY	NOT NULL	The unique identifier of the management entry
Admin_id	INT	FOREIGN		The identifier of the admin managing the entry. References to **Admin(Admin_id)**
Showtime_id	INT	FOREIGN		The identifier of the showtime being managed. References to **Showtime(Showtime_id)**
Manage_Date	TIMESTAMP		NOT NULL	The timestamp of the management action
Description	VARCHAR(10)			Description of the management action

Table 16: Information of **ShowtimeManagement** table

Field Name	Data Type	Key	Constraints	Meaning
Manage_id	INT	PRIMARY	NOT NULL	The unique identifier of the management entry
Admin_id	INT	FOREIGN		The identifier of the admin managing the entry. References to **Admin(Admin_id)**
Movie_id	INT	FOREIGN		The identifier of the movie being managed. References to **Movie(Movie_id)**
Manage_Date	TIMESTAMP		NOT NULL	The timestamp of the management action
Description	VARCHAR(10)			Description of the management action

Table 17: Information of **MovieManagement** table

Field Name	Data Type	Key	Constraints	Meaning
Manage_id	INT	PRIMARY	NOT NULL	The unique identifier of the management entry
Admin_id	INT	FOREIGN		The identifier of the admin managing the entry. References to **Admin(Admin_id)**
Voucher_id	INT	FOREIGN		The identifier of the voucher being managed. References to **Voucher(Voucher_id)**
Manage_Date	TIMESTAMP		NOT NULL	The timestamp of the management action
Description	VARCHAR(10)			Description of the management action

Table 18: Information of **VoucherManagement** table

4 Database Optimization

This section explores various techniques to optimize database performance and ensure efficient data management. Key approaches include the use of **indexes**, which improve query performance by enabling efficient data retrieval, and **views** or **materialized views**, which optimize query execution and simplify data representation. Additionally, **stored procedures** encapsulate business logic, enhancing maintainability, reducing redundancy, and minimizing network overhead. Finally, **triggers** automate specific actions in response to predefined database events, ensuring consistency and reducing manual intervention. Together, these methods contribute to a scalable and high-performing database environment.

4.1 Indexes

Indexes are a fundamental component of database optimization, designed to significantly enhance query performance, especially when dealing with large datasets. By implementing indexes on critical columns, the efficiency of operations such as filtering, sorting, and joining is greatly improved. This optimization reduces query execution time, allowing the database to handle complex queries more effectively and ensuring a better overall system performance.

The effectiveness of these indexes was tested using the **MTS-1M** dataset, which simulates a large-scale database environment, includes the following tables and corresponding record counts:

Table Name	**Record Count**
`BookingSeat`	1,000,000
`Booking`	1,000,000
`Showtime`	1,000,000
`ShowtimeManagement`	1,000,000
`Seat`	2,449,040
`User`	100,000
`Redemption`	1,000,000
`Room`	70,000
`Voucher`	10,000
`VoucherManagement`	10,000
`MovieGenre`	3,006
`Admin`	2,000
`Movie`	1,500
`Theater`	10,000
`MovieManagement`	20
`Genre`	10
`SeatType`	3

Table 19: Record counts for each table in the **MTS-1M** dataset

To further improve query performance, we applied 7 indexes on key tables. The specific indexes are as follows:

Index Name	**Indexed Fields**	**Table Name**	**Index Type**
`idx_booking_time`	`time`	`Booking`	B-Tree
`idx_booking_status_time`	`status, time`	`Booking`	B-Tree
`idx_theater_name`	`name`	`Theater`	B-Tree
`idx_theater_city`	`city`	`Theater`	B-Tree
`idx_showtime_date`	`date`	`Showtime`	B-Tree
`idx_movie_title`	`title`	`Movie`	B-Tree
`idx_redemption_date`	`redeem_date`	`Redemption`	B-Tree

Table 20: Indexes defined in our Movie Ticketing System

All the indexes mentioned above are implemented as **non-clustered index** using **B-tree indexes** due to their versatility and efficiency. B-trees are highly effective in handling equality comparisons, range queries (e.g., >, <, `BETWEEN`), and ordered traversals, making them suitable for diverse query types. Their balanced tree structure ensures consistent and predictable retrieval times, even as the dataset grows in size. Additionally, B-trees are particularly well-suited for common query patterns, as the indexed columns, such as dates, titles, and names, frequently appear in `WHERE`, `ORDER BY`, and `GROUP BY` clauses. These features make B-trees the optimal choice for enhancing the performance of our database schema.

Query	**Not using index**	**Using index**
Information about bookings from August 2024 to the end of 2024	199.208 ms	**131.963 ms**
Information about bookings with the status ’Cancelled’ from August to the end of December 2024	229.345 ms	**137.147 ms**

Table 21: Indexes in **Booking** table

Query	**Not using index**	**Using index**
List of movies screened during a specific time period (from 01/07/2024 to 21/07/2024)	178.331 ms	**142.793 ms**
Find the cinemas and showtimes where a specific movie with a given title starting with ’Inception’ is being screened during a specific period (from 09/06/2024 - 09/07/2024)	87.930 ms	**45.634 ms**

Table 22: Indexes in **Movie** and **Showtime** table

Query	**Not using index**	**Using index**
History of voucher redemptions during 08/09/2024 to 22/09/2024	58.053 ms	**21.737 ms**

Table 23: Indexes in **Redemption** table

Query	Not using index	Using index
Retrieve all movies screened at theaters with a specific name starts with ‘CGV’ during 01/08/2024 and 15/08/2024, including the theater name, movie details, and showtime schedule	114.068 ms	**48.323 ms**
Retrieve all movies screened in theaters located in cities start with ‘A’ from 01/03/2024 to 01/04/2024, including detailed information about the theater, screening room, movie, and showtime. The data is sorted by theater, screening date, and movie start time	160.947 ms	**82.902 ms**

Table 24: Indexes in **Theater** table

4.2 Views & Materialized Views

Views and materialized views simplify data access by providing a virtual table based on query results. Views dynamically retrieve data, ensuring up-to-date results, while materialized views store query results for faster access. They enhance query efficiency, improve data security by exposing only necessary information, and simplify complex queries through abstraction. In this subsection, we will discuss the views/materialized views used for the database.

4.2.1 MovieGenresInfo

This view provides detailed information about movies along with their associated genres by listing all movie attributes and consolidating genre names into a single field. A regular **view** is chosen because the dataset related to **movies** and **genres** is relatively small, making the performance benefits of a materialized view negligible. The simplicity of the query, with basic joins and aggregation, allows the database engine to handle it efficiently without the need for additional storage or maintenance overhead associated with a materialized view.

4.2.2 ShowtimeOccupancy

This view displays occupancy data for each showtime, including theater name, movie title, date, start time, total confirmed bookings, seats booked, room capacity, and occupancy rate. It is highly useful for analyzing theater performance and optimizing schedules or promotions.

A **materialized view** is chosen for **ShowtimeOccupancy** to handle the computational complexity of joining multiple tables, counting bookings, and calculating occupancy rates. These operations are resource-intensive on large datasets, but precomputing the results significantly improves query performance for read-heavy tasks like generating reports. Since showtime data changes only when bookings are updated, an **incremental refresh** is recommended, as it efficiently updates only the modified data without reprocessing the entire dataset.

Query	View Execution Time (ms)	Materialized View Execution Time (ms)
SELECT * FROM ShowtimeOccupancy;	5450.990	**232.783**

Table 25: Comparison of View vs. Materialized View Execution Time

4.2.3 TopBookingUsers

This view ranks users by the number of confirmed bookings they have made, including user details (ID, name, email) and total confirmed bookings, highlighting the most active and loyal customers. A **materialized view** is chosen because aggregating booking data for all users, especially in a large database, is computationally expensive. By precomputing this information, query performance improves significantly for reporting and frequent retrieval. An **incremental refresh** is preferred as data changes are event-driven (e.g., bookings, cancellations) and do not require recalculating the entire dataset.

Query	View Execution Time (ms)	Materialized View Execution Time (ms)
SELECT * FROM TopBookingUsers;	691.892	**17.209**

Table 26: Comparison of View vs. Materialized View Execution Time

4.2.4 DetailedBookingSummary

Offers a comprehensive summary of booking details, including user information, booking time and status, movie title, showtime schedule, theater details (name, city), seat information (row, number, type, price), sorted by the most recent bookings.

A **materialized view** is **not** chosen for **DetailedBookingSummary** because the data changes **frequently** due to new bookings, cancellations, or updates, requiring constant refreshing that negates its performance benefits. Additionally, the query relies on simple joins without complex aggregations, and proper indexing is sufficient to optimize its execution without the need for materialization.

4.2.5 MoviePerformance

This view summarizes movie performance by aggregating the total number of showtimes, confirmed bookings, and revenue generated for each movie or branch, sorted by revenue in descending order. A **materialized view** is chosen because the query involves multiple aggregations over potentially large datasets, which can be computationally expensive when executed frequently. Precomputing these results improves query efficiency for analytics and reporting. An **incremental refresh** is preferred, as it efficiently updates only the modified data (such as new bookings or showtimes) without reprocessing the entire dataset.

Query	View Execution Time (ms)	Materialized View Execution Time (ms)
SELECT * FROM MoviePerformance;	7570.300	**0.385**

Table 27: Comparison of View vs. Materialized View Execution Time

4.3 Stored Procedures

Stored procedures are precompiled SQL code blocks that execute specific tasks within a database. They improve performance by reducing query parsing, promote reusability, and enhance security by encapsulating business logic. Stored procedures also simplify complex operations by bundling multiple queries into a single callable unit. In this subsection, we will discuss the stored procedures used for the database.

4.3.1 InsertMovie

- **Description:** Adds a new movie to the `Movie` table.
- **Input parameters:**
  - `input_title` (VARCHAR): The title of the movie.
  - `input_description` (TEXT): A description of the movie.
  - `input_language` (VARCHAR): The language of the movie.
  - `input_rating` (DECIMAL): The movie rating (from 1 to 10).
  - `input_duration` (INT): The duration of the movie (in minutes).
  - `input_release_date` (DATE): The release date of the movie.
- **Result:** Inserts a new movie into the `Movie` table and returns the `Movie_id` of the new movie.

4.3.2 InsertRedemption

- **Description:** Adds a redemption record to the `Redemption` table.
- **Input parameters:**
  - `input_user_id` (INT): The ID of the user.
  - `input_voucher_id` (INT): The ID of the voucher.
- **Result:** Inserts a record into the `Redemption` table with the status `Available`.

4.3.3 InsertShowtime

- **Description:** Adds a movie showtime to the `Showtime` table.
- **Input parameters:**

  - input_start_time (TIME): The start time of the showtime.
  - input_end_time (TIME): The end time of the showtime.
  - input_date (DATE): The date of the showtime.
  - input_room_id (INT): The ID of the cinema room.
  - input_movie_id (INT): The ID of the movie.

- **Result:** Inserts a new showtime record into the Showtime table.

4.3.4 InsertUser

- **Description:** Adds a new user to the User table.
- **Input parameters:**
  - name (VARCHAR): The name of the user.
  - email (VARCHAR): The email of the user.
  - password (VARCHAR): The password of the user.
  - phone (VARCHAR, NULL): The phone number of the user.
  - address (VARCHAR, NULL): The address of the user.
  - date_joined (TIMESTAMP): The timestamp when the user joined.
  - dob (DATE, NULL): The date of birth of the user.
  - loyalty_points (INT): The number of loyalty points of the user.
- **Result:** Adds a new user to the User table.

4.3.5 InsertVoucher

- **Description:** Adds a new voucher to the Voucher table.
- **Input parameters:**
  - input_description (TEXT): The description of the voucher.
  - input_discount_percentage (INT): The discount percentage of the voucher.
  - input_expiry_date (TIMESTAMP): The expiry date of the voucher.
  - input_points_required (INT): The number of points required to redeem the voucher.
- **Result:** Adds a new voucher to the Voucher table and returns the Voucher_id of the voucher.

4.3.6 UpdateMovieByTitle

- **Description:** Updates the movie details based on the movie title.
- **Input parameters:**
  - `input_title` (VARCHAR): The title of the movie to update.
  - `input_description` (TEXT): The new description of the movie.
  - `input_rating` (DECIMAL): The new rating of the movie.
- **Result:** Updates the description and rating of the movie.

4.3.7 UpdateUserByEmail

- **Description:** Updates the user details based on email.
- **Input parameters:**
  - `emails` (VARCHAR): The email of the user to update.
  - `names` (VARCHAR, NULL): The new name of the user.
  - `passwords` (VARCHAR, NULL): The new password of the user.
  - `phones` (VARCHAR, NULL): The new phone number of the user.
  - `addresss` (VARCHAR, NULL): The new address of the user.
  - `dobs` (DATE, NULL): The new date of birth of the user.
- **Result:** Updates the user details based on email.

4.3.8 UpdateVoucher

- **Description:** Updates the voucher details.
- **Input parameters:**
  - `input_voucher_id` (INT): The ID of the voucher to update.
  - `input_description` (TEXT): The new description of the voucher.
  - `input_expiry_date` (TIMESTAMP): The new expiry date of the voucher.
  - `input_points_required` (INT): The new number of points required to redeem the voucher.
- **Result:** Updates the voucher details.

4.3.9 BookTickets

- **Description:** Books tickets for a user for a particular showtime.
- **Input parameters:**
  - `p_user_id` (INT): The ID of the user.
  - `p_showtime_id` (INT): The ID of the showtime.
  - `p_seat_ids` (INT[]): An array of selected seat IDs.
  - `p_voucher_id` (INT, NULL): The voucher ID if the user has one.
- **Result:** Creates a new booking record and associates the selected seats with the booking.

4.3.10 UpdateBookingStatus

- **Description:** Updates the booking status (Confirm or Cancel).
- **Input parameters:**
  - `p_booking_id` (INT): The ID of the booking to update.
  - `p_new_status` (VARCHAR): The new status of the booking.
- **Result:** Updates the booking status and calculates loyalty points for the user if the booking is confirmed.

4.3.11 SimulatePayment

- **Description:** Simulates the payment process for the booking.
- **Input parameters:**
  - `p_booking_id` (INT): The ID of the booking to pay for.
  - `p_payment_status` (BOOLEAN): The payment status (TRUE if successful, FALSE if failed).
- **Result:** Updates the booking status after payment is successful or failed.

4.4 Functions

Functions in SQL are blocks of code designed to perform specific tasks, typically processing and returning values based on input. There are two main types: **Built-in functions** and **User-defined functions**, which are created by users for custom operations. These functions optimize data queries, reduce code repetition, and enhance reusability within the database system. Functions offer key benefits, including improved performance, automation of repetitive tasks, enhanced code readability, and consistent, secure data processing through encapsulated logic. With functions, SQL not only retrieves data but also processes it in a flexible and effective way. This subsection consists of the functions that our group used for the database.

4.4.1 CalculateBookingPrice

Description: Calculates the total price of all seats in a specific booking.
Input:

- `input_booking_id` (INT): The ID of the booking.

Output:

- Total price (INT) of all seats in the booking.

4.4.2 CalculateRevenueByDay

Description: Calculates the revenue from all bookings on a specific day.
Input:

- `input_date` (DATE): The date for which the revenue is calculated.

Output:

- Total revenue (INT) on the specified day.

4.4.3 CalculateRevenueByMonth

Description: Calculates the revenue from all bookings in a specific month and year.
Input:

- `input_month` (INT): The month for which the revenue is calculated.
- `input_year` (INT): The year for which the revenue is calculated.

Output:

- Total revenue (INT) in the specified month and year.

4.4.4 CalculateRevenueByYear

Description: Calculates the revenue from all bookings in a specific year.
Input:

- `input_year` (INT): The year for which the revenue is calculated.

Output:

- Total revenue (INT) in the specified year.

4.4.5 FindTheatersByName

Description: Finds theaters based on their names.
Input:

- `input_theater_name` (VARCHAR): Partial or full name of the theater.

Output:

- A table with the following columns:
  - `Theater_id` (INT): Theater ID.
  - `Name` (VARCHAR): Theater name.
  - `Address` (VARCHAR): Theater address.
  - `City` (VARCHAR): City where the theater is located.

4.4.6 FindMoviesByTheater

Description: Finds movies available in a specific theater.
Input:

- `input_theater_id` (INT): The ID of the theater.

Output:

- A table with the following columns:
  - `Movie_id` (INT): Movie ID.
  - `Title` (VARCHAR): Movie title.
  - `Description` (TEXT): Movie description.
  - `Language` (VARCHAR): Language of the movie.
  - `Rating` (DECIMAL): Movie rating.
  - `Duration` (INT): Movie duration in minutes.
  - `Release_Date` (DATE): Movie release date.

4.4.7 FindSeatsByBooking

Description: Retrieves seat information for a specific booking.
Input:

- `input_booking_id` (INT): The ID of the booking.

Output:

- A table with the following columns:
  - `Seat_id` (INT): Seat ID.
  - `Row` (VARCHAR): Seat row.
  - `Number` (INT): Seat number.
  - `Seattype_id` (INT): ID of the seat type.

4.4.8 FindAvailableSeatsByShowtime

Description: Finds available seats for a given showtime, excluding seats that are already confirmed or pending.

Input:

- `input_showtime_id` (INT): The ID of the showtime.

Output:

- A table with the following columns:
  - `Seat_id` (INT): Seat ID.
  - `Row` (VARCHAR): Seat row.
  - `Number` (INT): Seat number.
  - `Seattype_id` (INT): Seat type ID.
  - `Seattype_name` (VARCHAR): Name of the seat type.
  - `Price` (INT): Price of the seat.

4.4.9 FindShowtimesByMovieAndTheater

Description: Retrieves showtimes for a specific movie in a given theater.

Input:

- `input_movie_id` (INT): The ID of the movie.
- `input_theater_id` (INT): The ID of the theater.

Output:

- A table with the following columns:
  - `Showtime_id` (INT): Showtime ID.
  - `Start_Time` (TIME): Showtime start time.
  - `End_Time` (TIME): Showtime end time.
  - `Date` (DATE): Showtime date.

4.4.10 FindTheatersByMovieId

Description: Finds theaters showing a specific movie, along with the total number of rooms in each theater.

Input:

- `input_movie_id` (INT): The ID of the movie.

Output:

- A table with the following columns:
  - `Theater_id` (INT): Theater ID.
  - `Theater_Name` (VARCHAR): Theater name.
  - `Address` (VARCHAR): Theater address.
  - `City` (VARCHAR): City where the theater is located.
  - `Total_Rooms` (INT): Total number of rooms in the theater.

4.4.11 update_expired_vouchers

Description: Updates the status of expired vouchers to "Expired" in the redemption table.
Input: No input parameters.
Output:

- A message indicating that expired vouchers have been updated in the redemption table.

4.4.12 view_redemption_vouchers

Description: Displays a list of vouchers redeemed by a specific user, along with their current status and redemption date.
Input:

- `userid` (INT): The ID of the user.

Output:

- A table with the following columns:
  - `Voucher_id` (INT): Voucher ID.
  - `Status` (VARCHAR): Current status of the voucher.
  - `Redeem_Date` (TIMESTAMP): Date and time the voucher was redeemed.

4.4.13 get_booking_info

Description: Retrieves detailed booking information including user details, theater, movie, seat details, and total price.
Input:

- `booking_id_param` (INT): The ID of the booking.

Output:

- A table with the following columns:
  - `user_name` (VARCHAR): Name of the user who made the booking.
  - `theater_name` (VARCHAR): Name of the theater.
  - `room_name` (VARCHAR): Name of the room where the movie is shown.
  - `movie_title` (VARCHAR): Title of the movie.
  - `start_time` (TIME): Showtime start time.
  - `end_time` (TIME): Showtime end time.
  - `seats` (TEXT): List of booked seats.
  - `total_price` (INT): Total price for the booking.
  - `loyalty_points` (INT): Loyalty points earned for the booking.

4.4.14 CalculateUserMonthlySpending

Description: Calculates the total amount spent by a user on movie bookings within a specific month and year.

Input:

- `UserId` (INT): The ID of the user.
- `Month` (INT): The month of interest.
- `Year` (INT): The year of interest.

Output:

- Total amount spent (NUMERIC) by the user in the specified month and year.

4.4.15 CalculateUserYearlySpending

Description: Calculates the total amount spent by a user on movie bookings within a specific year.

Input:

- `UserId` (INT): The ID of the user.
- `Year` (INT): The year of interest.

Output:

- Total amount spent (NUMERIC) by the user in the specified year.

4.4.16 CountUserMoviesWatchedInMonth

Description: Counts the number of distinct movies watched by a user in a specific month and year.

Input:

- `UserId` (INT): The ID of the user.
- `Month` (INT): The month of interest.
- `Year` (INT): The year of interest.

Output:

- The number of distinct movies (INT) watched by the user.

4.4.17 CountUserMoviesWatchedInYear

Description: Counts the number of distinct movies watched by a user in a specific year.

Input:

- `UserId` (INT): The ID of the user.
- `Year` (INT): The year of interest.

Output:

- The number of distinct movies (INT) watched by the user.

4.5 Triggers

Triggers are powerful database mechanisms that automate predefined tasks, ensuring data consistency and enforcing business rules. By responding to specific events such as inserts, updates, or deletes, triggers help maintain the integrity of the database while reducing manual intervention and potential errors.

4.5.1 deduct_loyalty_points_on_redemption

This trigger is executed **after a new redemption is inserted** into the `Redemption` table, provided the status of the redemption is marked as `Available`. It automates the process of deducting loyalty points from a user's account when a voucher is redeemed. The trigger checks if the user has enough loyalty points to redeem the voucher. If sufficient points are available:

- The required points are deducted from the user's account.
- A success message is displayed, detailing the transaction.

If the user lacks sufficient points, the redemption is aborted, and an error is raised.

4.5.2 check_conflicting_showtime

This trigger is executed **before inserting a new showtime** into the `Showtime` table. It ensures there are no scheduling conflicts in the same room by checking for overlaps in screening times. The trigger prevents double bookings by verifying:

- If the `Room_id` and `Date` of the new showtime conflict with any existing showtime.

Result: If a conflict is detected, the insertion is aborted, and an error message is raised, indicating the conflict.

4.5.3 check_duplicate_movie

This trigger is executed **before inserting a new movie** into the `Movie` table. It ensures that the database does not contain duplicate entries for the same movie. The trigger checks:

- If a movie with the same `Title` (case-insensitive), `Language`, and `Release_Date` already exists in the database.

Result: If a duplicate is found, the insertion is aborted, and an error message is raised to indicate the duplicate entry.