MOVIE TICKET BOOKING SYSTEM (DBS_PR_09)

Hrithik Raj Gupta (2019B2A70995P)

Rajan Sahu (2019B4A70572P)

Table of Contents

1.System Requirement Specifications (SRS)

1.1 Database and SQL Files

The given project aims to capture the main ideas of handling a movie ticket reservation system. The following documentation serves to provide sufficient guidelines to follow while testing and evaluating the project. The attached code consists of a single SQL file containing all the required queries, data definition, procedures, functions etc. The order in which the components of the SQL file has been kept is the same as mentioned in the project guidelines of the course **CS F212 Database Systems and Concepts, Fall Semester 2022**.

1.2 Scope of the Project

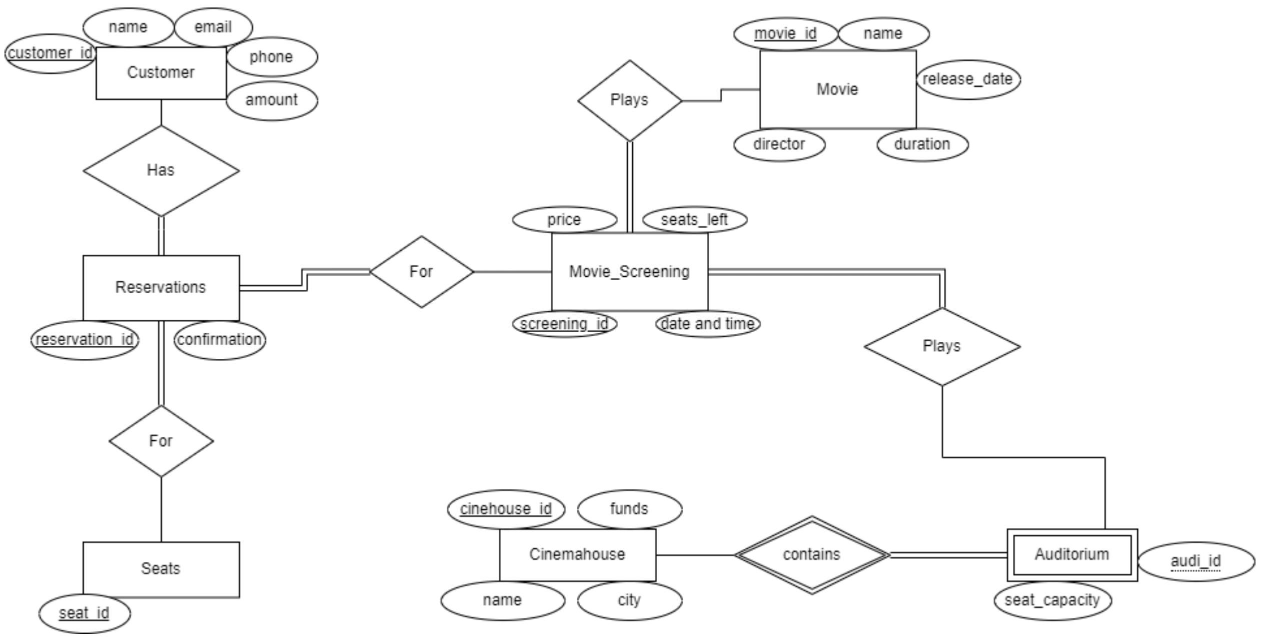
The scope of the project is to simply capture the basic requirements of a real-life simulation of an online movie ticket reservation system. The requirements that the following system fulfils is that of storing the important data in a non-redundant manner. The intricacies of the database such as normalization and schemas has been discussed ahead in the next section. Since an online reservation system is accessible to multiple users, this necessitates the use of transactions to take care of consistency of the database. Hence all the queries as per the requirements and intuition derived from real life scenarios has been handled using a suitable mode that is either concurrently handled or not. The system that we have designed is also meant to be quite compatible with any frontend framework and thus could be easily expanded according to the needs of the user.

1.3 Model Assumptions

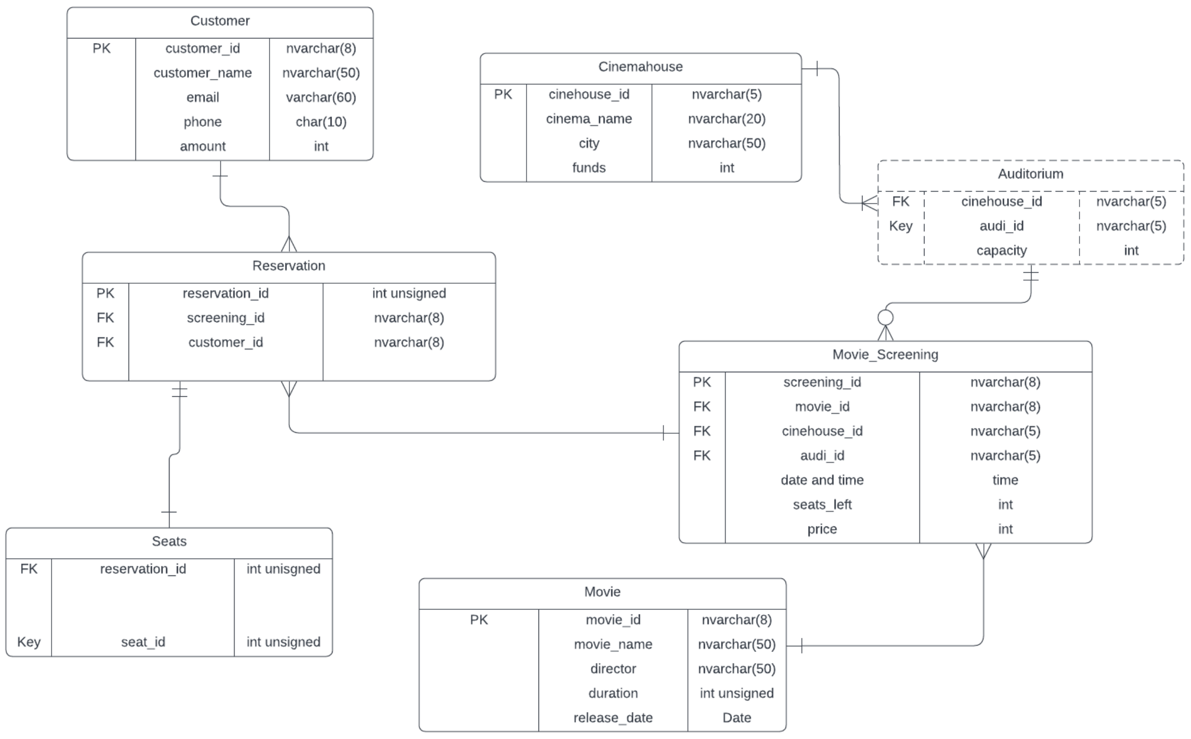
For the given model we considered taking limited number of seats in a theatre, 10 to be exact. Also since there is no GUI or frontend implemented in fulfilment of the project, so we decided to stick with a fixed layout for all the theatres. This also helped us in making the schemas more redundant free and suitable, yet necessary amount of information was retrievable through normal SQL queries. Further assumptions about the cardinality of relations have been explained in detail in the following section. Sufficient attention has been paid to relations between different entities associated with the system.

2. System Modelling

2.1 ER Diagram



2.2 Schema Design



Customer

PK	customer_id	nvarchar(8)
	customer_name	nvarchar(50)
	email	varchar(60)
	phone	char(10)
	amount	int

Cinemahouse

PK	cinehouse_id	nvarchar(5)
	cinema_name	nvarchar(20)
	city	nvarchar(50)
	funds	int

Auditorium

FK	cinehouse_id	nvarchar(5)
Key	audi_id	nvarchar(5)
	capacity	int

Reservation

PK	reservation_id	int unsigned
FK	screening_id	nvarchar(8)
FK	customer_id	nvarchar(8)

Movie_Screening

PK	screening_id	nvarchar(8)
FK	movie_id	nvarchar(8)
FK	cinehouse_id	nvarchar(5)
FK	audi_id	nvarchar(5)
	date and time	time
	seats_left	int
	price	int

Seats

FK	reservation_id	int unisgned
Key	seat_id	int unsigned

Movie

PK	movie_id	nvarchar(8)
	movie_name	nvarchar(50)
	director	nvarchar(50)
	duration	int unsigned
	release_date	Date

2.3 Data Normalization and Design

As mentioned earlier in the documentation, this project serves to fulfil the basic requirements of a movie ticket reservation database system. This is easily compatible with any suitable frontend and can be expanded. Coming to the data model assumptions first, we have a **Customer** database consisting of the necessary details to distinguish a customer entity. Followed by that we have a **Movie** database that stores all the current movies that are being hosted at present, a **Cinemahouse** database that captures all the theatres nearby (or in specific cities). Every **Cinemahouse** has multiple **Auditoriums**, however an auditorium turns out to be a weak entity so the relation between Cinemahouse and Auditorium is a weak entity relation as auditoriums themselves have unique identity in a single theatre, but different theatres can have same identification scheme for numbering their auditoriums. Next, we have **Movie_Screening** database which stores all the shows for different movies, at different times, date, locations etc. And lastly **Reservations** database stores the respective reservations made for a particular movie_screening, for a particular seat and for a particular customer.

The schema design given in the previous page shows the list of tables that have been finally considered for creating the database. All the tables have atomic attributes, ensuring all the tables are in 1NF. Apart from that for 2NF we need to check whether there is any partial dependency, that is a non-prime attribute being functionally dependent on parts of candidate key. The individual entity sets already have single primary keys so there is nothing to worry about the tables not being in 2NF. If we check relation tables such as **Reservations**, **Movie_Screening** which involves multiple foreign keys however there exists a unique primary key in every table, thus 2NF has been naturally enforced during the design of the table. Talking about transitive dependencies there are none as no data is redundant in a single table that is there are no transitive dependencies. One particular aspect of Multivalued Dependencies was handled while implementing the seats. Since technically speaking an auditorium can have multiple seats, and multiple seats could have been repeated for each theatre, therefore we thought it was better to keep it simple and store a reservation_id with a seat_id since through a reservation_id sufficient information about the seat's occupancy can be derived through other tables.

2.4 List of Tables required

Apart from the tables mentioned in the schema design, we felt that there wasn't any specific requirement of other tables. Since we had the constraint of ensuring normalizations, so using the tables that has been proposed by our design might prove to be bit slow during too much of data to process through various joins etc. However, we saw a trade-off between data redundancy and speed. Since the basic motto was former so we went with it. If someone wishes to improve the speed, then many other tables can be created such as prepare tables for every theatre and then store all the seats and then easily retrieve data about occupancy instead of the way we have implemented.

2.5 Additional Components

We decided to use transactions inside procedures for implementing the booking and cancellation of seats. Other than the basic queries, since our database design enforced us to use mostly joins and matching on several keys, so retrieving a particular view included many subquery statements. Hence, we felt it necessary to keep the code organized and even in case the project gets expanded by implementing a frontend, it would be more convenient to call procedures instead of selecting multiple lines of queries.

For simple retrievals we just have used simple queries in the SQL file itself. Other important demonstrations and explanations about the code functionality has been mentioned in the code document and explained in detail through the video.