PROJECT REPORT

1. Please list out changes in the directions of your project if the final project is different from your original proposal (based on your stage 1 proposal submission).

The primary objective of our project was to construct a web application for a cinema ticket booking system. This primary subgoal was successfully achieved. Additionally, as promised, we delivered on two the vital components of a user side portal and an admin-side management system. Additionally, when we developed the project, we realized that the schema and database overview specified in the first stage wasn't complex enough to capture the entire essence of our application. Hence, with each stage, we have continued to expand and refine the logical schema of our application. With that being said, one component we couldn't create was an email confirmation of ticket booking with a QR code. Furthermore, we built an admin-side portal that gives a high-level overview but didn't have the time to design an interactive statistical dashboard for the admins to analyze and visualize their sales based on movies and genres to better understand the customer base. With that being said, we added a significant new component to our application based on the suggestion of our TA during the mid-term demo i.e., we successfully implemented a movie-recommendation system that recommends similar movies to users who book a ticket.

2. Discuss what you think your application achieved or failed to achieve regarding its usefulness.

Our cinema ticket booking system achieved a visually appealing user experience that enables users to book tickets similar to our initial prototype design. User experience is a big motivator which drives customers to stick to a platform and keep them coming back. With this particular ideology in mind, we were assiduous in designing the ticket booking and the user profile webpage. The application also allows two stakeholders, i.e., admin and user to login into the system which were initially defined. The users are able to register, log in, browse, book tickets for the movies. We took the implementation a step further and designed a working search bar. The search bar can search the database for hundreds of movies that are currently showing and display the relevant titles based on pattern-matching. We also take meticulous care in building the front-end that displays the result of the search operation. Additionally, we take this implementation a step further and provide the users with the opportunity to delete bookings.

On the other hand, the admin is able to add, update and delete movies, users, and shows in the application (achieving all the CRUD operations for both stakeholders). Apart from this, we were also able to implement a stored procedure that is used to return recommended movies based on current bookings and two triggers in our application. We also achieved a creative component through this, by implementing a recommendation system that recommends the users a movie they can watch based on their previous preferences using content-based filtering. We look at the textual description of the plot of the current movie and compare it with the textual description of every other movie in the database using a popular distance (cosine distance) as a measure of similarity. The stored procedure is used to fetch any and every movie that has a similarity score within a pre-specified threshold and suggest the closest movie. One more important security feature implemented in our application is the storage of passwords in the database in an encrypted manner

(using MD5 hashing). Thus, we were able to achieve an application that is highly relevant to a real-world scenario of movie ticket booking.

3. Discuss if you changed the schema or source of the data for your application

The schema proposed in the project proposal consisted of 5 tables - User, Movie, Shows, Bookings, and Seats. These tables were implemented to the dot as planned in the first draft. However, certain attributes were added to these tables to capture additional detail that we felt is required in order to do justice to the application. While progressing through the iterations of the development, we realized some minor logical inconsistencies in our original plan and had to think of creative ways for reconciling such disparities.

To cite an example, we originally had a seatId attribute in the Bookings relation that would store the seat number associated with the booking. However, we soon realized that in the practical world, one person would book the seats for an entire group, and hence one booking would ideally have 4-5 seat numbers associated with it. So, we overcame this challenge by adding the bookingId as a foreign key referencing the Booking relation from the Seats relation. This allowed us to get around this conundrum and let multiple seats be associated with a single booking Id. There were some more logical problems that needed to be resolved. To quote another instance, earlier, we had also modeled the Seat table to be a weak entity on the Booking entity, but later had to change it to be a weak entity on the Shows entity.

The key source of our data was the list of movies that we acquired from Rotten Tomatoes. We made sure that we stored the movies with the same movieId that was used to store them by Rotten Tomatoes. This gave us the added advantages of being able to write simple web crawlers that can use the Rotten Tomato URL along with the movie id and query the appropriate webpage to fetch additional information like the movie cast, the movie post, or the number of Emmy nominations, etc. For user details, and other secondary data, we custom-developed random data generators in Python to create synthetic data that is highly similar to the real world. We also work with significant amounts of data, having over 2k movies, 2.5k users, over 6k shows and 10k+ bookings.

4. Discuss what you change to your ER diagram and/or your table implementations. What are some differences between the original design and the final design? Why? What do you think is a more suitable design?

For our project, we had to make a few substantial changes in the schema to make it fully functional and cover various features as mentioned above. Most changes came when we started thinking about how to run advanced queries that would be actually helpful and add value to our application. When we came up with a few ideas, we realized that we needed to make sure that we improve the connectivity between our databases so that we can properly join different relations and write powerful queries. Firstly, we had to add movieId as a foreign key in the ‘Shows’ table and we had to add showId and bookingId in the ‘Seats’ Table so that we could add the seat visualization feature. This allowed us to write a query to get the status of individual seats in a show and change their color based on their status. This change also enabled us to show the number of available seats for each show, which very relevant to what one might see on a real-world movie booking web-application. We also had some heated, but healthy discussions on how the admins’ functionality should work, and more importantly how we should map that relation cardinality in the UML

representation. Another change would be the addition of a recommendation relation in the UML. This relation is currently absent from the proposed logical design as it was designed and developed post the mid-term demo.

5. Discuss what functionalities you added or removed. Why?

We were not able to implement the components which we had proposed earlier in our project proposal mentioned above as we were advised to add a recommendation system to our application. So, we added a creative component of recommendation where our system understands the user's likes and recommends the user movies based on their previous booking which otherwise, they might have not discovered on their own. We felt this was an interesting and highly useful feature to include to improve customer experience and satisfaction which could increase the number of people who use our application. We look at the textual description of the plot of the current movie and compare it with the textual description of every other movie in the database using a popular distance (cosine distance) as a measure of similarity. This computation is done using Python and its NLTK library as it has inbuilt features for tokenizing, vectorizing and building the TF-IF Matrix. Additionally, if the recommendation is actually relevant, people might actually book the tickets for the recommended movies which would in turn increase the profits for the application.

In turn for adding this functionality, we had to remove time from the development of another important feature. Earlier we had the plan of sending the booking details to the users when they booked a ticket. This would include a QR code that had embedded information about the movie, show, and seat. Additionally, we would have sent a calendar invite that the users could have used to add the show timing to their google or outlook calendar. We can also send an email to the users as a reminder 60 minutes before the booking. We also originally planned to have an advanced query that would look at the advanced booking of all shows from the next week and tell the admins about the most, and the least profitable movies. Using this information, the admins could take actionable interventions like cancelling shows of least profitable movies and allocating those shows to the best-selling movies. If we had more time, we would have loved to add these two functionalities to our system as well.

6. Explain how you think your advanced database programs complement your application.

Automation is the 8th wonder of the world. It makes the life of the developers and IT system managers very easy. We use trigger to automate two important parts in our application. The first one is the creation of show details. Whenever an administrator runs the query for adding a new show for a movie, we use an event-based trigger for adding twenty-five seats for this show with availability attribute set to 1. This saves a lot of time and work in manually configuring every seat for every show that gets added. We also run an additional trigger that takes into account system design constraints like the amount of storage we would like to spend to keep running our applications. Presently, we have implemented a trigger to drop any movie shows and the bookings and seats associated with it after 6 months. This helps in storing only the most relevant data and minimizing the storage required by our system.

We also use a stored procedure for recommending movies to users. The stored procedure uses a movieId of every booking made by a user and then fetches every movie that has a similarity score above a threshold. The stored procedure then runs a loop to keep a track of the details that is the

most similar to the bookings made by the user. The stored procedure is developed in such a way that it makes it very easy to change its functionality for retrieving the top 3, or the top 5 similar movies instead of a single movie title.

7. Each team member should describe one technical challenge that the team encountered. This should be sufficiently detailed such that another future team could use this as helpful advice if they were to start a similar project or where to maintain your project.

One of the main barriers in our implementation was the technical challenge to call our stored procedure to return a recommended movie inside a while loop which would return all the user's bookings. This led to failure in calling the stored procedure from inside the loop as the output of the first SQL query to display all the current bookings was storing the result in a buffer and returning them one after another inside the loop. People familiar with JavaScript would know that different calls to the Database do execute simultaneously and this creates a problem when the code is being executed sequentially. Typically, the async keyword is used in Java Script to handle this conundrum and make multiple asynchronous calls to the database system at the same time. We were successfully in solving this problem by using mysqli_free_result($first_result) and mysqli_next_result($db) to free the buffer and make a successful stored procedure call.

One of the key technical challenges which we came across during working on the project was hosting the PHP application on the Google Cloud Platform. While hosting the SQL data was much easier, there was not a lot of documentation available for hosting PHP applications on the Google Cloud Platform. As a potential solution, to anyone trying to make a similar application in the future, we suggest the use of a more modern tech stack, like React, as it will have significant documents, blog post, and video tutorials available on the internet for free.

Another challenge we faced was getting our hands on practical data. While there are many sources like Rotten Tomatoes and IMDB for getting movie details, there is a serious paucity for practical data about real world customers. A potential solution could be using data from public national surveys. We took an alternate approach. We wrote several scripts which generated randomized data for bookings, seat availability, shows, and users. However, we can't just use nested for-loops for randomly generating different pieces of data. Here, the data needed to be consistent across all tables which was a challenge for us and took a lot of work. We wrote the random generators in a manner that they build on one another. We generate users and associate them movies. Then using this corpus of data, we design a random generator to mimic the show timings of a real-world cinema complex. After having the shows, we write another random data generator to make bookings for these shows. These bookings had to be consistent on two ends, the users and movies. For example, the same user shouldn't have a booking for two different movies at the same time and one show must not be overbooked. We even took care to make sure that the bookings are similar to expected behavior of users in real world (for example, a user seldom books a single ticket, more tickets are booked for movies in prime-time slots, etc.).

Another big wall we ran into was with storing the poster images for all the movies in our database. This didn't follow a standard data type and was causing inconsistencies in our database. We later found out BLOB as an alternative data type which was made available through SQL server which helped us store and retrieve the appropriate poster, stored locally, from the database. An additional

tip for someone going down the similar route is to store the movies with the exact-same movieId that the original source of their data used. This has a hidden advantage of being able to write sophisticated web crawlers that can use the source-data URL along with the movie id and query the appropriate webpage to visit and download the movie posters in a single place. This will prove to be a lifesaver and save 'HOURS' in implementation.

Lastly, as we learnt in the lectures, Normalizing the schemas is not a one-stop solution to all the problems. I think we tried to over-normalize our database schema during the development stage. In doing so, we removed almost all the redundancy from our relations. This has two significant effects. First, writing queries became difficult and challenging as we needed to join a greater number of relations to achieve relatively simpler tasks, and second, this made our system less efficient. As we understood in the lectures that joins are very costly and reduce the efficiency of the system.

8. Are there other things that changed comparing the final application with the original proposal?

The original proposal did not cover movie recommendations based on a user's current booking history. We were able to accomplish this feature in our application to provide a creative component to the project and move our application more in the direction of a real-world booking website.

9. Describe future work that you think, other than the interface, that the application can improve on

I think we tried to over-normalize our database schema during the development stage. In doing so, we removed almost all the redundancy from our relations. This increased the number of joins required in our system to run powerful and advanced queries. As and when our system expands and has more users and movies, we need to think about the scalability of the system. Having a database schema that minimizes joins or stores the result of the most used joins as temporary tables might be important to deal with scalability challenges.

Next, our system does not have any support for handling concurrent requests at the moment. Suppose only one ticket is left for a very trending movie, and multiple users try to book it at the same time. Such a scenario is not handled by our system. We can improve the consistency of our application through the use of isolation levels to avoid such mishaps from ever taking place.

Furthermore, we can think about maintaining a separate NoSQL database like Neo4j to store and map the relations between different users and the movies that they book. This component of the data is highly intertwined and could be visualized using Cypher querying language. This allows us to move from content based filtering to a hybrid approach (content + collaborative filtering) and improve the efficiency of our inhouse recommendation system, leading to greater profits for the company.

An additional line of thought would be using a gamified system to increase the activity and headcount of people actively using the application. Currently, the most popular systems in the market are utilizing gamification to introduce loyalty programs. We can implement these features

through the use of triggers. Consider a very simple use-case. We can store the birthday of the consumers and write a trigger to send a 10% discount to every customer in their birthday month.

Lastly, as we said, our system used two triggers and we plan to expand the functionality by adding more triggers. So, it might also be useful to consider when the triggers fire. Instead of firing the show dropping trigger at 10:00am in the morning (when the system is the most active) we should have the least active time slots (like, 4:00 am) for firing the triggers. These are some high-level considerations that will help us take our cinema booking application to the next level.

10. Describe the final division of labor and how well you managed teamwork.

The work was equally divided among all the team members. Team members were assigned various tasks which were somewhat familiar to them but challenging at the same time so that they could fortify their base in the respective domains. All four team members worked on the database and advanced tasks like advanced queries. The stored procedure and triggers were implemented in groups of two while the creation and population of the database were also divided equally among everyone. The front end was mainly done by two group members and the other two members focused on integrating the front end with database by adding SQL queries in the PHP framework. Lastly, the work of every member was reviewed by the entire team that created additional channels for catching silly mistakes, providing constructive criticism, and teamwork.