Middle East Technical University

STAT311, Modern Database Systems

A Database-Driven Movie Watch Party Platform

By

Pelin İşgör	**Burak Şahin Küçük**	**Fajrin Akbarli**	**Ahmet Turaç**
2510337	2614758	2656072	2666790

Ankara | Dec 28, 2025

Contents

Abstract

This project presents metucorn, a comprehensive, database-driven movie watch party platform developed as part of the STAT311 Modern Database Systems course. The system demonstrates database concepts through a web application built using PostgreSQL (via Supabase), Next.js 14 and React 18. The database architecture consists of 14 tables that manage the complex relationships between users, movies, tickets, viewing parties, and ratings. The application supports user authentication (email/password and OAuth), movie catalog browsing, ticket purchasing, creating and joining synchronized viewing parties, movie rating and commenting, viewing history tracking, and a comprehensive admin panel for content management and analysis. The entire system was deployed using a zero-cost architecture: Vercel for front-end hosting and Supabase for a managed PostgreSQL database, demonstrating real-world cloud deployment applications.

1 | Introduction

The rise of streaming platforms and increased online interaction between people created a demand for sharing the movie experience beyond cinemas. However, this demand could not be met by streaming platforms. They have largely avoided adding social integration to their systems and continue to do so. In addition, they have brought cinemas to the brink of extinction, and people can no longer socialize even in cinemas. Most theaters are no longer filling up. We thought of replacing these streaming services, which greatly prevent people from coming together over films, with our shared movie viewing platform. This way, people will be able to watch movies with their friends while watching a movie online. In addition, they will be able to use the tickets they purchase through the system in physical cinema environments with QR codes on a one-time basis. Thus, with MetuCorn, we aim to increase interaction between people in both streaming and traditional movie viewing environments.

1.1 | How We Come In?

To clearly illustrate the need for a system like ours, we use a Venn diagram.

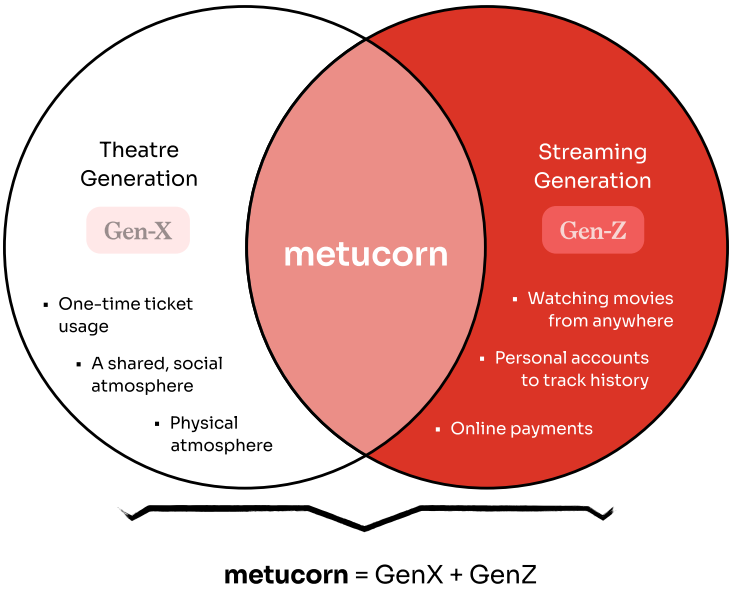

Figure 1.1: Venn Diagram to Conclude How we Come In

As shown in Figure 1.1, diagram compares traditional cinema/theater experiences with online streaming platforms and highlights how our project integrates key features of both.

On the left side of the diagram, traditional cinema and theater experiences are represented. These systems are primarily characterized by:

- Watching movies in a physical location, such as a cinema or theater.
- Single-use ticketing, where each ticket allows one viewing.
- A shared and social atmosphere, where audiences experience content together in real time.

On the right side of the diagram, online streaming services are shown. These platforms typically emphasize:

- The ability to watch movies from any location using internet-connected devices.
- Personal user accounts with individualized access.
- Online payment systems and stored user viewing history.

The overlapping area in the centre of the Venn diagram represents our project. This is where our system offers the advantages of both approaches. We have the convenience and accessibility of online streaming, combined with keeping it event-based and social like regular cinema. And users can watch movies online, join in for watch parties with their friends and use single-use digital tickets, like real cinema tickets, to limit access to content. Combining both models, our system provides no necessity to choose or prioritize offline cinema and online streaming. Rather, it offers a single source solution for ease, social interaction and access control as one system. This unification is what makes our project unique, compared to other alternatives, and required.

1.2 | Objectives

The primary aim of the project is to create a database system for an online cinema platform. The purpose is to provide the core support for digital movie content management and user interaction in a compact and efficient way.
The primary objectives of this project were to:

- Identifying the problem and thinking about the full solution.
- Deciding about technologies to solve the problem.
- Design a normalized relational database schema with ER diagram.
- Integrate the database with a modern web application.
- Deploy the end-to-end system to cloud infrastructure to gain users.

To sum it up, the project shows how you could develop a movie website through basic database operations; that is, modelling data for your entities, connecting them together and running simple queries on the collection of records. The focus is on building a reliable and well-structured database system rather than on advanced multimedia or streaming technologies

1.3 | Project Scope

We have identified certain features to ensure that our system works perfectly and serves users well:

- User authentication and profile management with Supabase's auth system.
- Movie catalog with detailed information (director, cast, genres, ratings).
- Ticket purchasing system with mock payment processing (no real data retrieved).
- Watch party creation, joining, and management.
- QR code (one time physical ticket) generation for watching movie in the physical theaters.
- Mock movies with custom react video player which uses embedded Vimeo contents.
- Rating and review system ($1 - 10$ scale with text reviews).
- Comprehensive admin panel for content management.
- Analytics dashboard for administrators.

1.4 | Technology Stack

The main scope of the course is PostgreSQL 15, so we used PostgreSQL and pgAdmin for our local works. PostgreSQL is industry-standard relational database management system. When we go through the application, we choose supabase, a cloud platform that provides postgresql support, authentication, and auto-generated REST API. If we used Adminer + FastAPI (or any other service such as Django, Java Spring, etc.), it would take much more time compared to 'Supabase Basic' because Supabase also created authentication service, RESTful API from database schema and row level security. In frontend layer we used Next.js 14 because this React framework has App router for server-side rendering. React version was 18.3.1 and for styling we used Tailwind CSS 3.4.15. For designing the frontend layer and our additional promotion things such as logo[1], presentation, etc. we used Figma.

Additional Libraries:

- *react-player 2.16.0*: For custom video player in website.
- *qrcode.react 4.2.0* : QR-code generation for given ticket string e.g. TKT-XXXX.
- *recharts 3.6.0*: Charts and data visualization for admin analytics.
- *lucide-react 0.462.0*: Totally free and open-sourced icon library.

For deployment we used Vercel for Frontend hosting when we commit to our github repository Vercel directly changes existed production deployment to a new one. For database integration we used built-in Supabase Cloud.

1.5 | Final Application Overview

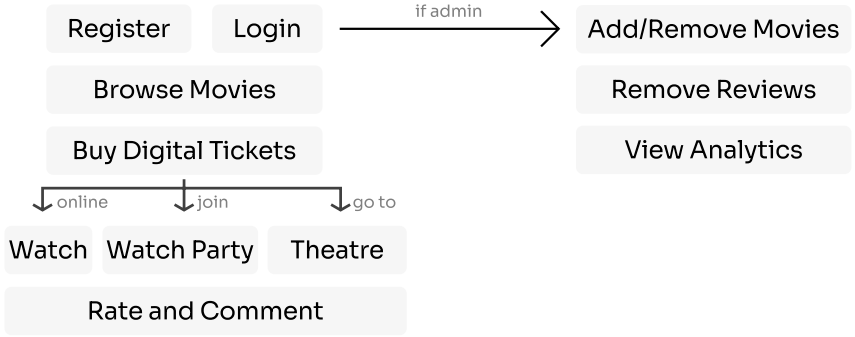

Figure 1.2: Venn Diagram to Conclude How we Come In

In Figure 1.2 illustrates the basic schema what you can do in the our finalized application. The primary objective of this project is to design and implement a database system for an online cinema platform. The system aims to support the core functionalities required for managing digital movie content and user interactions in an organized and efficient manner. Specifically, the project seeks to:

- User authentication system that ensures each user will unique and different account.
- Enable users to browse available movies, purchase digital tickets, watch movies online, and submit ratingsafter viewing.
- Store and manage essential information related to users, movies, tickets, payments, actors, and movie genres.
- Provide administrative capabilities that allow system administrators to add, update, or delete movie records, ensuring that the movie catalog remains current and well-maintained.
- Generate basic analytical results, such as identifying the most popular movies and highest-rated movies, based on user activity and feedback.

2 | Database Schema and ER Diagram

2.1 | Overview of ER Diagram

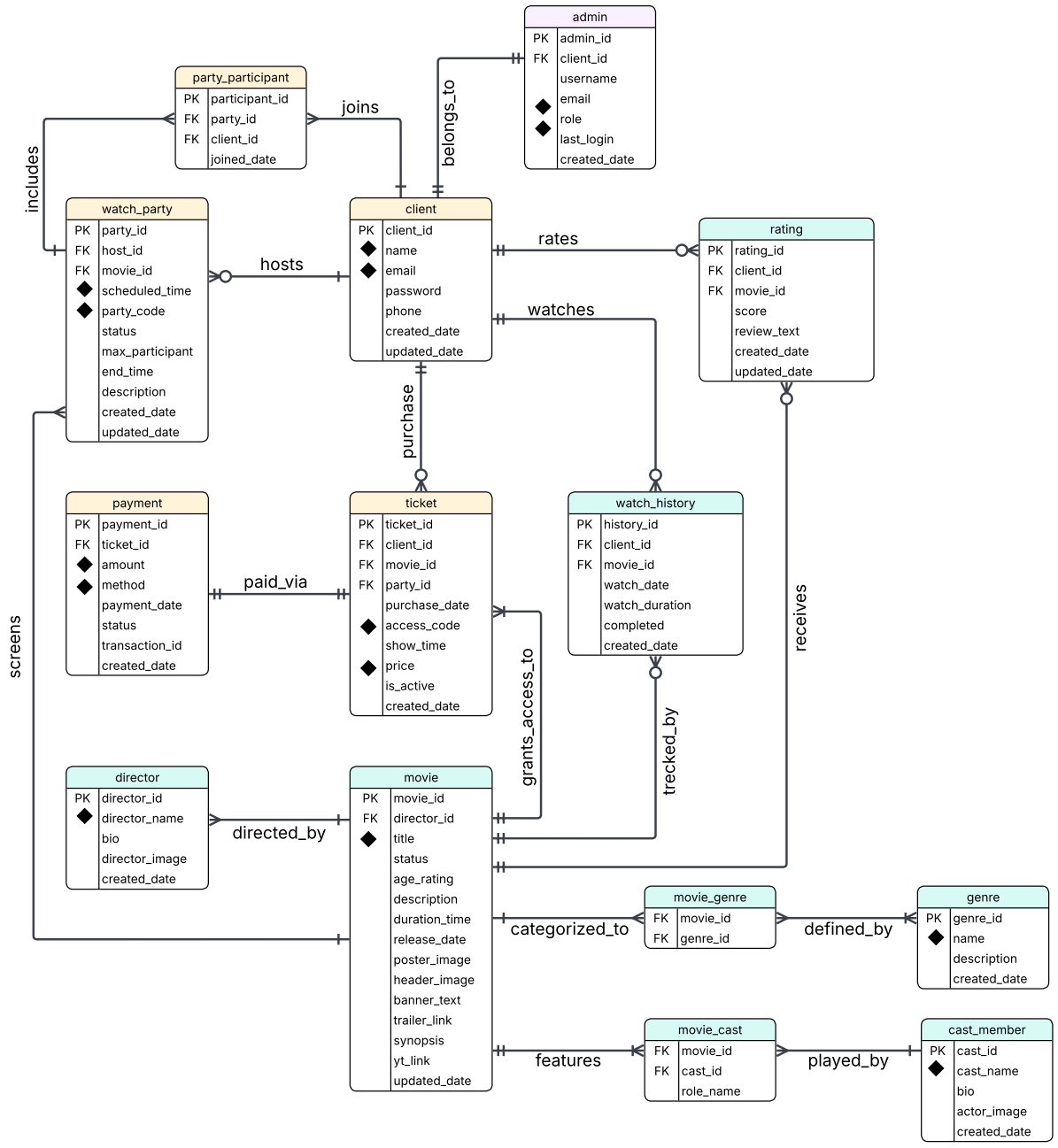

Figure 2.1: Complete ER Diagram from Lucid Chart

We created full ER Diagram as shown in the Figure 2.1. The Entity-Relationship (ER) diagram for the MetuCorn Cinema project represents the database structure of an online movie booking and viewing platform. The database is designed with a relational approach that aims to maintain data consistency while supporting diverse user interactions. Normalization rules were followed during the design process to reduce data redundancy and allow for future expansion of the schema.

Registered users in the system are represented by the Client entity, and this table forms the basis of most interactions on the platform. Each user is uniquely identified by the `client_id`, which is used as the primary key. Basic user data such as name, email, password, and contact information are stored in

this table. Because users participate in movie purchasing, viewing, and rating processes, the Client table is referenced by several tables such as Ticket, Rating, `Watch_History`, and Payment. Administrator users are modeled separately under the Admin entity. Connecting the Admin table to the Client table with a foreign key prevents the mixing of normal user data with administrative permissions, providing a clearer authorization structure.

Information about movies is stored in the Movie entity. This table contains basic information about the film, such as title, duration, release date, age rating, and viewing links. Each film is linked to the Director table via the `director_id` foreign key. This structure allows the same director to be associated with multiple films, and director information can be stored without duplication. This approach conforms to normalization principles and facilitates data updates.

Films are classified by genre using the Genre entity. Because a film can belong to multiple genres, and each genre can contain multiple films, there is a many-to-many relationship between Movie and Genre. This relationship is modeled using the `movie_genre` intermediary table. Similarly, actor information is stored in the `Cast_Member` table, and many-to-many relationships with films are established through the `movie_cast` table. This intermediary table also contains information about the actors' roles in the film.

The operational processes in the system are primarily carried out through the Ticket entity. A ticket represents a user's access to a specific film and, in some cases, participation in social viewing events. The Ticket table is linked to the Client and Movie tables using foreign keys and contains information such as purchase date, screening time, fee, and access status. Payment information is stored separately in the Payment table. The Payment table is linked to the Ticket table and contains information such as payment method, amount, transaction number, and status. This structure ensures traceability of financial transactions and supports basic audit requirements.

The social tracking feature is modeled with the `Watch_Party` entity. This table allows users to create scheduled viewing events for a specific movie. Each viewing party is linked to a host user and a movie. Participants are tracked via the `party_participant` table. This separation allows participation numbers and interaction data to be managed without complicating the ticketing structure.

To analyze user behavior, the `Watch_History` and Rating entities are also included in the database. The `Watch_History` table records when and for how long a user watched a movie, as well as whether the viewing was completed. The Rating table contains user ratings and optional comments for movies. Both tables are linked to the Client and Movie tables using foreign keys.

Overall, the ER diagram reflects a modular and organized database design. Primary keys uniquely identify each record, while foreign keys maintain integrity between tables. Separate modeling of intermediate tables and functional components used for many-to-many relationships ensures the system consistently supports both operational processes and analytical queries.

2.2 | Relationship Types and Cardinality

The following relationships exist in the MetuCorn Cinema database, designed to ensure data integrity and normalization:

2.2.1 | One-to-One (1:1) Relationships:

- `ticket` 1 → 1 `payment` (Each ticket is associated with exactly one unique payment transaction)
- `client` 1 → 1 `admin` (An admin profile belongs to a specific unique client/user)

2.2.2 | One-to-Many (1:N) Relationships:

- `director` 1 → N `movie` (One director can direct many movies, but a movie has one director)
- `client` 1 → N `ticket` (One client can purchase many tickets)
- `client` 1 → N `watch_party` (One client can host multiple watch parties)
- `client` 1 → N `rating` (One client can write many reviews/ratings)

- `client` 1 → N `watch_history` (One client has many watch history records)
- `movie` 1 → N `ticket` (One movie can have many tickets sold for it)
- `movie` 1 → N `watch_party` (One movie can be screened in many different watch parties)
- `movie` 1 → N `watch_history` (One movie can be watched by many users)
- `movie` 1 → N `rating` (One movie can receive many ratings)
- `watch_party` 1 → N `ticket` (One party can have multiple tickets associated with it)

2.2.3 | Many-to-Many (N:M) Relationships:

In our database, Many-to-Many relationships are handled by "Junction Tables" to satisfy normalization rules:

- `movie` ↔ `genre`: A movie can belong to multiple genres, and a genre can contain multiple movies. Handled by: `movie_genre` table.
- `movie` ↔ `cast_member`: A movie features multiple actors, and an actor can play in multiple movies. Handled by: `movie_cast` table (includes `role_name` attribute).
- `client` ↔ `watch_party` (Participation): A client can join multiple parties, and a party can have multiple participants.
  Handled by: `party_participant` table (includes `joined_date` attribute).

2.3 | Design Decisions and Key Constraints

When creating the ER diagram, not only were entities and relationships defined, but various design decisions were also made to strengthen data integrity. Primary keys were used in each table to ensure that records were uniquely identified. Relationships between tables were established through foreign keys, preventing invalid references.

In addition, some logical constraints were applied at the database level. For example, associating each ticket record with only a single payment transaction prevents access rights from being granted without a completed payment. Similarly, uniqueness constraints prevent a user from rating the same movie multiple times or attending the same viewing party again. This approach ensures that the data validation burden is not left solely to the application layer but is also supported by the database.

2.4 | Support for Analytical Queries

The generated ER diagram not only supported the operational functioning of the system but also enabled the execution of analytical and statistical queries. Separating ticket and payment data from user interaction data allowed us to obtain more meaningful and consistent results in the analysis processes.

For example, by evaluating viewing history and user ratings together, we were able to perform an analysis of the relationship between movie viewing rates and user satisfaction. Furthermore, modeling the relationships between movie genre and movie actor through intermediate tables enabled the efficient execution of complex queries such as identifying the most popular genres, comparing the performance of films starring specific actors, or conducting genre-based revenue analyses.

2.5 | Scalability and Extensibility Considerations

The modular structure of the ER diagram provides an infrastructure suitable for future system expansion. The absence of tight dependencies between existing tables facilitates the addition of new features. For example, additional functions such as a subscription-based usage model or personalized recommendation systems can be integrated into the system by adding new entities without modifying existing core tables.

Furthermore, thanks to the structure designed in accordance with normalization rules, data redundancy is prevented and performance losses are reduced. This increases the sustainability and manageability of the database, both in the event of an increase in the number of users and the system becoming more complex.

3 | Database and SQL Implementation

In previous sections, we have talked about the current streaming climate and how there is this constant debate of streaming vs theatre going on. We proposed our idea of a streaming platform that would cater to both sides of the argument and with our ER diagram already shown, we are now ready to continue with the implementation of these ideas. In the end, ideas don't amount to much when they stay as just ideas. This section will be about the SQL implementations of them, demonstrating how we created a database that suits our needs and how we handled relations that were discussed in the previous section.

Other than the junction tables we created, most tables are pretty straightforward. There are a total of 14 tables as shown in the ER diagram, 3 of them being junction tables between movie-genre, movie-cast and watch_party-client. Below are the codes for movie table creation:

```sql
-- movie table
CREATE TABLE IF NOT EXISTS movie (
    movie_id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    director_id UUID REFERENCES director(director_id) ON DELETE SET NULL,
    title VARCHAR(500) NOT NULL,
    status VARCHAR(50) DEFAULT 'available',
    age_rating VARCHAR(10),
    description TEXT,
    synopsis TEXT,
    duration_time INTEGER, -- in minutes
    release_date DATE,
    poster_image TEXT,
    header_image TEXT,
    banner_text TEXT,
    trailer_link TEXT,
    yt_link TEXT,
    price DECIMAL(10, 2) DEFAULT 0.00,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP
);
```

As seen, we used optimal data types for each attribute. V ARCHAR is great for storing varied, rather small string sizes since it doesn't add spaces to fill the specified length. Using V ARCHAR for attributes such as the title, status and age_rating ensures that we don't waste any memory, since these columns will only occupy the space they need to. The limit of each V ARCHAR also acts as a constraint since there surely is a problem if the title of a movie is longer than 500 characters, for instance. On the other hand, for attributes such as the description, synopsis and trailer_link, we used TEXT since they will hold much longer texts compared to our attributes with data type VARCHAR. The attribute duration_time takes the duration of a movie as an integer value in minutes, created_at and updated_at keeps the logs of changes to a movie with precise time values using TIMESTAMP. Just as all the other IDs in this database, movie_id and director_id belong to the data type UUID. Using UUID instead of simply INTEGER is a more secure approach because the IDs are abstracted away. Had it been the case that integer values were used, someone with malicious intent could easily find the number of movies, directors or more importantly clients we have by just looking at the IDs that are user1, user2, user3 and so on, for instance. Using UUID, we don't have this problem because they are practically untraceable. The final data type to mention is the DECIMAL, which is fitting for price values. Here, the price can be up to 10 digits and we show 2 decimal points, which is more than enough considering a streaming platform.

An important thing to mention here is the concepts of primary and foreign keys. In the line where we define movie_id, you can see that the "PRIMARY KEY" keyword was used. This means that our movie_id uniquely defines the movie table and is unique. Just below that line, the keyword "REFERENCES" creates a foreign key to the director table to get the director_id from that table. This line ends with ON DELETE SET NULL, which means that in the case where a director is deleted from a director table (director_id is the primary key for the director table) and its director_id is gone, the director_id column of the movie table will be set to null. The keywords DEFAULT and NOT NULL also deserve a mention. The former lets us pick a default value in the case nothing explicit is entered as a value to the attribute, latter is a constraint which says that particular space can not be left empty.

We will now follow this with the creation of the genre table:

```
-- genre table
CREATE TABLE IF NOT EXISTS genre (
    genre_id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    name VARCHAR(100) UNIQUE NOT NULL,
    description TEXT,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP
);
```

Since it holds much less data than what the movie table will hold, the codes for the genre table are much simpler. We again initiate the table with the keyword CREATE TABLE IF NOT EXISTS (IF NOT EXISTS prevents duplicates if we already have a table named genre). As we did for all IDs, genre_id also of the type UUID, created with the 4th generation uuid generator function. Genre name has to be unique, which is forced by the UNIQUE keyword. It also can not be null, a similar constraint was present for the movie title just above. A movie without a title or a genre instance without the actual name renders other attributes meaningless as the reader would approve. The last two lines were also present in the movie table, with the exact same data types. We hold the description of a genre with data type TEXT, and also keep the time of creation by using precise timestamp information given by built-in SQL function CURRENT_TIMESTAMP.

The SQL code contains many other tables and most of them use the same concepts that are shown with the movie and genre tables. For instance, below are the tables for our clients and watch parties which they will create:

```
-- client table
CREATE TABLE IF NOT EXISTS client (
    client_id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    user_id UUID,
    name VARCHAR(255) NOT NULL,
    email VARCHAR(255) UNIQUE NOT NULL,
    phone VARCHAR(50),
    avatar_url TEXT,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP
);
```

```
-- watch party table
CREATE TABLE IF NOT EXISTS watch_party (
    party_id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    host_id UUID REFERENCES client(client_id) ON DELETE CASCADE,
    movie_id UUID REFERENCES movie(movie_id) ON DELETE CASCADE,
    scheduled_time TIMESTAMP WITH TIME ZONE NOT NULL,
    party_code VARCHAR(20) UNIQUE NOT NULL,
    status party_status DEFAULT 'scheduled',
    max_participants INTEGER DEFAULT 50,
    end_time TIMESTAMP WITH TIME ZONE,
    description TEXT,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP
);
```

Same constraints that were shown with the movie and genre tables are also present here; such the usage of UNIQUE and NOT NULL for the email attribute of the client table and the party_code attribute of the watch_party table. A careful reader might have noticed that there is something extra going on with the status party_status line. To explain that line, let's first see the custom types we created, which party_status being one of them:

```sql
-- custom types
DROP TYPE IF EXISTS party_status CASCADE;
DROP TYPE IF EXISTS payment_status CASCADE;
DROP TYPE IF EXISTS payment_method CASCADE;

CREATE TYPE party_status AS ENUM ('scheduled', 'active', 'completed', 'cancelled');
CREATE TYPE payment_status AS ENUM ('pending', 'completed', 'failed', 'refunded');
CREATE TYPE payment_method AS ENUM ('credit_card', 'debit_card', 'paypal', 'wallet');
```

We created 3 separate custom types; called party_status, payment_status and payment_method. These types assume restricted values that are specified using the AS ENUM keyword. The attribute status from the table watch_party is of type party_status, and its default value is set to be "scheduled". These custom types are useful because in real life you have limited options for their values. Most services offer you several but limited options to pay your bill, for instance. Having these predefined set of options for these attributes makes for a tidier organization because unlike other attributes their values will always be one of these predefined options. You may have a hundred different movie titles but the same is not applicable for these 3 attributes. You can also see the utilization of data type payment_status and payment_method below, in the payment table:

```sql
-- payment table
CREATE TABLE IF NOT EXISTS payment (
    payment_id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    ticket_id UUID REFERENCES ticket(ticket_id) ON DELETE CASCADE,
    amount DECIMAL(10, 2) NOT NULL,
    method payment_method NOT NULL,
    payment_date TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
    status payment_status DEFAULT 'pending',
    transaction_id VARCHAR(255),
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP
);
```

No default value for the attribute "method" is selected, but it is specified as not being null as seen. The default value for the status is "pending".

A rather more interesting piece of code is the implementation of tables that handle many to many relationships, such as the one between movie and genre. Obviously, a movie can belong to multiple genres, and a genre can be given to multiple movies. The movie "Snatch" for example, is both a crime and comedy movie. "Reservoir Dogs" is also a crime movie, so we see the relationship clearly. Below is the initiation of the junction table movie_genre:

```sql
-- movie-genre junction table
CREATE TABLE IF NOT EXISTS movie_genre (
    movie_id UUID REFERENCES movie(movie_id) ON DELETE CASCADE,
    genre_id UUID REFERENCES genre(genre_id) ON DELETE CASCADE,
    PRIMARY KEY (movie_id, genre_id)
);
```

As expected, movie_id and genre_id attributes take their value from the tables movie and genre, respectively, by creating a foreign key to the parent tables by using the keyword REFERENCES. The 2-tuple (movie_id, genre_id) is the primary key of this junction table and it uniquely identifies each record in the table. This pair has to be unique of course, one of the main benefits of this implementation is going to be very apparent in one of our queries in the next section. This unique pair will enable us to handle queries that depend heavily on many to many relations being understood and implemented correctly.

Before moving onto the next section, it is worth mentioning the process of creating a mock sample data that reflects real life data. The reason that such sample data should reflect real life scenarios is because when you generate completely "random" data in projects with similar scope such as ours, you will be ignoring the particular way the data is "formed" in real life. For instance, your clients cannot

possibly create a watch party before creating an account, or purchasing at least one movie. There are so many factors that contribute to the data, people watch movies at a certain time period in a day, they don't purchase your entire catalogue of movies, their behaviours are tied to many factors that can't be accounted for by just running a randomized script that populates your data.

The rather static part of our data was created manually. We handpicked 30 movies and got all the information about them from the internet, such as the director of the movie, actors, their roles, age rating, descriptions etc. For our website which will be discussed in more detail in the coming parts, we needed movie posters and images along with trailers for each movie. We got the posters and backdrop images from moviedb[2], and trailers from YouTube. Text based information like short bios for directors & actors and descriptions of movies was taken from IMDB website and Wikipedia, mostly. We populated the respective tables using this information. The codes for data insertion are not presented here since they are huge blocks of texts, but the curious reader can refer to the end section where we will provide all the codes used in this project. The data insertion section is mostly boring INSERT INTO statements though, not particularly fun. In the end, we had 30 genres, 17 directors, 54 actors and 30 movies.

Returning to the handling of sample data concerning mostly the client logs (how many movies they bought & watched, when they watched them, how high they rated it, when they rated it), we used a logical function where we have set certain restrictions so that the data produced by it will resemble what a real version of a streaming platform would produce. Our restrictions ensured several things, some of which were as follows:

1. There were no time travellers, all dates and timestamps were consistent with each other.
2. Most traffic occurred between the hours 8pm-11pm, which is the usual "prime time" in real life.
3. Movie ratings were consistent with their real life ratings. We have set restrictions to lower bounds and upper bounds and produced the results between these bounds, sometimes skewed to one side.
4. We have created arrays of names and surnames and created 500 users from the combinations of them (which at some point resulted in us having Ali Koç as a customer).

Since this part is not the focus of this project and in general our course, we will not be discussing the specifics of sample generation. Once again, the curious reader can refer to the end section to see the SQL codes. Now that we talked about how various good practices and concepts were standardly carried out in our SQL codes, this section is coming to an end. The next section will focus on our research questions and queries and will further give insights about SQL concepts and their implementations.

4 | Queries Related with BI and Analytics

In this section, four research questions and their SQL queries along with their outputs and explanations will be given. These four questions are the same from the presentation, but they will be much more in-depth here, since there is no time constraint. The queries and our comments on them cover both BI and analytics related information. For each question, codes with explanations will be given, which will be followed by the output and interpretation of the results.

4.1 | Research Questions:

4.1.1 | What part of the day do most people watch movies?

This question is an important one because in modern life a day is structured in a way that certain activities will be done at certain parts of the day and watching a movie is no exception. This is why the term "prime time" exists in broadcasting, and in our case, streaming. SQL codes are as follows:

```
SELECT
    CASE
        WHEN EXTRACT(HOUR FROM watch_date) BETWEEN 5 AND 11 THEN 'Morning (05-11)'
        WHEN EXTRACT(HOUR FROM watch_date) BETWEEN 12 AND 19 THEN 'Afternoon/Early (12-19)'
```

```
        WHEN EXTRACT(HOUR FROM watch_date) BETWEEN 20 AND 23 THEN 'PRIME TIME (20-23)'
        ELSE 'Night/Late (00-04)'
    END AS time_of_day,
    COUNT(*) AS total_streams,
    ROUND(COUNT(*) * 100.0 / SUM(COUNT(*)) OVER(), 1) || '%' AS traffic_share
FROM watch_history
GROUP BY time_of_day
ORDER BY total_streams DESC;
```

Here, we select our data of interest from the table watch_history. Inside the SELECT statement, we use the conditional expression CASE WHEN-THEN END. This flows very naturally, we have 4 cases which differ from each other by their intervals that separate the day into 4 parts. By using the EXTRACT function we get the hour data from the attribute watch_date and if it is between particular bounds we use the THEN keyword to assign that interval a label. The line starting with ELSE accounts for the hours falling out of the first three cases. After finishing the query with the END keyword, we assign this categorization to a column named time_of_day by using the keyword AS. We use COUNT(*) function to count every row for each time category we just created, which will be shown as total_streams in the output. The following line is a bit more intricate. The function SUM(COUNT(*)) OVER() is called a window function, it calculates the grand total of all rows without collapsing the individual categories. Notice that this part is in the denominator (nominator is the counted total of each category), the end result of this line will be the rounded value of the fraction in terms of percentages, which is exactly what we want. With this the SELECT statement finally ends. Following lines indicate we choose all this from the table watch_history, we group by time_of_day which holds the result of our CASE-END expression. Finally we order by total_streams for an easier and more natural interpretation. Now let's see the output of our code in Figure 4.1:

	time_of_day text	total_streams bigint	traffic_share text
1	PRIME TIME (20-23)	2525	68.5%
2	Night/Late (00-04)	537	14.6%
3	Afternoon/Early (12-1...	336	9.1%
4	Morning (05-11)	288	7.8%

Figure 4.1: SQL output for Question 1 in pgAdmin

Earlier we said that our sample data simulates real life scenarios, this result shows that it indeed does. 68.5% of the traffic share occurs between 8pm-11pm, which is our prime time. Morning hours are much less crowded, with only 7.8% percent traffic being present during those hours. This information is especially critical in our case, since MetuCorn is at its core an online platform first. Our services are presented online, which means we will be managing servers. One thing everyone knows about servers is that they need maintenance. Having this information at hand is very valuable because it lets us form a maintenance schedule that does not hinder client enjoyment.

4.1.2 | What are the highest rated drama movies?

This question may seem rather trivial, but people love Top Rated lists and it could also help us in extending our catalogue in the future by letting us know what our customers love. In the end, the customer is king. SQL codes are as follows:

```
SELECT
    m.title,
    ROUND(AVG(r.score), 2) AS average_score,
    COUNT(r.rating_id) AS total_reviews
FROM movie m
JOIN movie_genre mg ON m.movie_id = mg.movie_id
JOIN genre g ON mg.genre_id = g.genre_id
JOIN rating r ON m.movie_id = r.movie_id
```

```
WHERE g.name = 'Drama'
GROUP BY m.movie_id, m.title
ORDER BY average_score DESC, total_reviews DESC
LIMIT 5;
```

Here, we select from the movie table using the alias ”m”. We have a multi-table join here, the movie table is joined to the junction table movie_genre by the column movie_id, and joined to the genre table by the column genre_id. The last join is to the rating table, which is again joined by the movie_id column. What we do here is by using the movie table as our anchor point, we navigate through multiple tables to get the data we want, using the junction table movie_genre as a connection point. We then only list the movies with genre ”Drama”, group them by movie_id and title, order them by their average score and limit the result to 5 rows to get a Top 5 list. Note that the job of counting all the ratings and calculating the average is done inside the SELECT statement at the top. The A VG aggregate function is used to get the mean rating score, which is then rounded to its two decimal digit form. Notice also how handy the foreign key concept comes off here, the statement ON m.movie_id = r.movie_id is a condition of the join task, it checks if the movie_id from the movie table matches the movie_id from the rating table. Since the rating table gets the movie_id from the movie table with the REFERENCES keyword, we ensure that it will return corresponding records. Without further ado, let us see the output:

	title character varying (500)	average_score numeric	total_reviews bigint
1	Once Upon a Time in Hollywo...	8.26	85
2	Gangs of New York	8.23	100
3	Oppenheimer	8.21	107
4	Whiplash	8.15	104
5	There Will Be Blood	8.14	103

Figure 4.2: SQL output for Question 2 in pgAdmin

As our columns, we see our selected values: title, average_score and total_reviews. Once Upon a Time in Hollywood comes out at top with an average score of 8.26 over 10 with a total of 85 reviews. It is followed closely by Gangs of New York with an average score of 8.23 points with 100 reviews. Total review count is important since scores settle to their actual representative values with more and more people reviewing them. Here though, they have very similar review counts so we don’t need to worry about it. Aside from showing the audience’s exquisite taste, this could give us siterunners an idea of what our audience is expecting from us regarding the new drama movies.

4.1.3 | Which users have spent the most money on the platform?

This is purely for identifying our most loyal customers. The reason for such a question could be that we may want to offer exclusive deals or bonuses to customers based on their support to us to further consolidate our relationship with them. SQL codes for this query are as follows:

```
SELECT
    c.name,
    COUNT(t.ticket_id) AS total_tickets_bought,
    SUM(t.price) AS total_revenue
FROM client c
JOIN ticket t ON c.client_id = t.client_id
GROUP BY c.client_id, c.name
ORDER BY total_revenue DESC
LIMIT 5;
```

The logic is very similar to the previous query, except this time we get the sum of all ticket prices a customer bought to get the total amount of money the client has spent. We count the tickets by their ids and assign it to a column named ”total_tickets_bought” then sum the ticket prices and assign them to a column named ”total_revenue”. Here we have one join, which is between the client table and the ticket

table, using aliases c and t respectively. They are joined by their matching client_id. Remember that client_id is a foreign key of the ticket table. By relating the clients with the tickets we ensure that each ticket sale is correctly attributed to a specific client. We group by the client_id and client name (client names are not unique, though the ids are) and order by the total_revenue to get our most loyal customers on top of the table. We limit the list to 5 rows. Following is the output table:

	name character varying (255)	total_tickets_bought bigint	total_revenue numeric
1	Umut Toprak	22	287.80
2	Buse Toprak	22	281.80
3	Tolga Keskin	22	277.80
4	Tuğçe Yıldız	22	274.80
5	Tuğçe Yılmaz	22	273.79

Figure 4.3: SQL output for Question 3 in pgAdmin

All of our top 5 spenders have bought 22 tickets, which is expected for us because while generating the sample we set the max value for the number of tickets a customer bought to 22 since we have only 30 movies. To keep it grounded that is, because a customer buying your entire catalogue is not likely, however much one could wish that. Since our ticket prices vary a lot by small amounts though, total revenue each client brought to us differs by small margins. Table shows Umut Toprak is our most loyal customer, bringing us a total of $287.80 in revenue, followed closely by Buse Toprak with $281.80 spent on tickets.

4.1.4 | Which genre has grossed the most money?

This question, at first glance, seems very simple and the question itself is indeed, simple. The answer to this question would be very helpful to help us extend our catalogue further, just like the second question. It would give us insights about which type of movies we should invest in and bring to the platform. Getting the answer to this question is a bit trickier compared to the previous queries though. We know from previous parts that movies and genres have a many to many relationship. This means that if we were to group movies by their genres and get the total amount earned from each genre, we would have counted the same movie twice, thrice or even more. For instance, when a movie is both a comedy and a drama, you will count that sale towards both genres, with the ticket's full price. This will artificially inflate the revenue gathered from each genre, which is not of use. The correct way to handle this would be to count each genre's effect on the revenue of a movie separately. Take the case of a movie sold with a price tag of $10 and let it be from the genres comedy and drama. If done right, this will result in comedy and drama genres gaining $5 each separately. Implementing this in the code is what separates this query from the previous ones. We make use of Common Table Expressions to solve the double counting issue:

```sql
WITH movie_genre_counts AS (
    SELECT movie_id, COUNT(*) as genre_count
    FROM movie_genre
    GROUP BY movie_id
),
genre_sales AS (
    SELECT
        mg.genre_id,
        SUM(t.price / mgc.genre_count) as adjusted_revenue
    FROM ticket t
    JOIN movie_genre mg ON t.movie_id = mg.movie_id
    JOIN movie_genre_counts mgc ON t.movie_id = mgc.movie_id
    GROUP BY mg.genre_id
)
SELECT
    g.name AS genre_name,
    ROUND(gs.adjusted_revenue, 2) AS total_genre_revenue
FROM genre g
JOIN genre_sales gs ON g.genre_id = gs.genre_id
```

```
ORDER BY total_genre_revenue DESC;
```

Common Table Expressions are useful when we need to separate a query into multiple parts to stay organized. CTEs hold the result of a set of queries temporarily as tables and enable us to reference them later. First CTE is declared with the keywords WITH-AS, named movie_genre_counts. Inside this block, we select movies by their ids from the junction table movie_genre and count how many genres they have, and assign it to an alias named genre_count, which will be referenced later. We group by movie_id to see how many genres each movie belongs to.

Second CTE starts with the line "genre_sales AS", and this is where the math happens. The first JOIN line joins the ticket table to the movie_genre table while the second links the tickets to the movie_genre_counts table we created with the first CTE. The former lets us get the ticket sales for each genre while the latter enables us to use the information from the first CTE, which is the genre count for each movie. We select the genre_id and the sum of ticket price divided by the genre count for the corresponding movie and label it adjusted revenue. This ensures we count the effect of each genre on the revenue.

The SELECT statement at the last part is where we show the final result. The genre table is joined with the genre_sales which is the result of our second CTE block to use the adjusted_revenue data. We select from this result and list the genre name and the total adjusted revenue (which is rounded to two decimal places), with columns named genre_name and total_adjusted_revenue. The output is below:

	genre_name character varying (100)	total_genre_revenue numeric
1	Drama	16492.97
2	Crime	12918.75
3	Comedy	5958.00
4	Action	5754.52
5	Thriller	5726.39
6	Adventure	3915.08
7	Fantasy	3503.32
8	Western	2739.32
9	Historical	2616.72
10	Sci-Fi	2332.84
11	Mystery	2244.68
12	Musical	1723.28
13	War	1601.41

Figure 4.4: SQL output for Question 4 in pgAdmin

Firstly, one can notice that the numbers are consistent with our dataset, consisting of movies that range in price from 5 to 25 dollars and a total user base of 500 people. Had it been the case that we didn't account for each genre's effect on the sales separately, these values would be much higher, possibly in the hundred thousand region, which would not be expected considering the scope of our sample. Moving on with our findings, not surprisingly, drama is the leading genre with a total revenue of $16492.97. Crime gets the second place with a total revenue of $12918.75. Drama alone has grossed almost as much money as the next three genres; comedy, action and thriller. By examining our client's preferences we can work on our catalogue and choose future movies accordingly.

In this section we employed several SQL concepts to write queries for our research questions, which gave

us insights about customer behaviour and preferences. Using the data gained from asking and answering such questions, our service can curate user experiences that leave both our customers and by extension us, pleased. We mark the end of this section.

5 | Fully Responsive Web Application

As we talked in introduction section, we completed our program with creating web application. This web application is still reachable in metucorn.tech.

5.1 | User Authentication

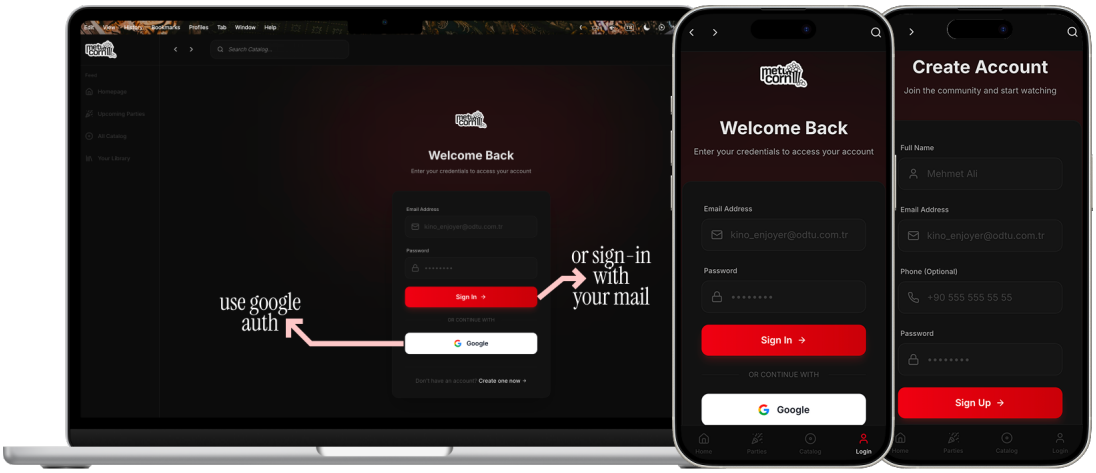

Figure 5.1: Log-in and Sign-up in Web Application of MetuCorn

Most of the time, implementing secure authentication system is not easy task. Thanks to Supabase Auth we directly solved this problem. Supabase integrates our client table with its own authentication ecosystem. This makes the authentication process effortless. For further details you can check Supabase Auth Docs. In our system we have responsive login and signup screens in Figure 5.1 that ensures people to register our systems via google or manual form system.

5.2 | Movie Browsing

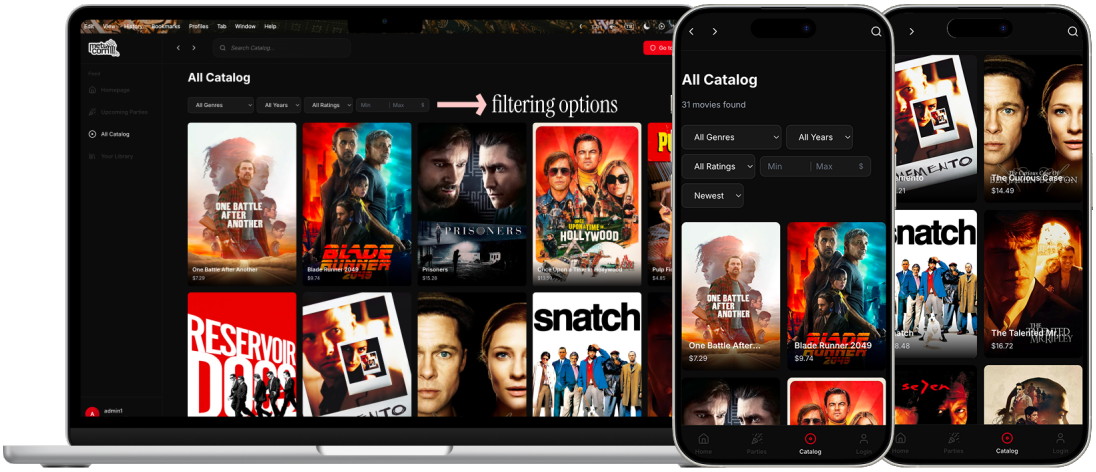

Figure 5.2: Movie Catalog in Web Application of MetuCorn

In Figure 5.2 we can see our catalog page. In this page people basically filter the movies. When a user selects a filter (e.g., genre or year), the MovieFilters component on the client side adds this value to the

URL as a query parameter (e.g., `/movies?genre=action&year=2023`). Next.js detects this change in the URL and re-renders the page. The server-side MoviesPage component reads these parameters via `searchParams`; it executes the query by adding filters such as `.eq()`, `.gte()`, `.lte()`, etc., as appropriate for the Supabase query.

5.3 | Movie Details

In movie details page we have user reviews, average rating, buying and watching options. And each movie has hero image.

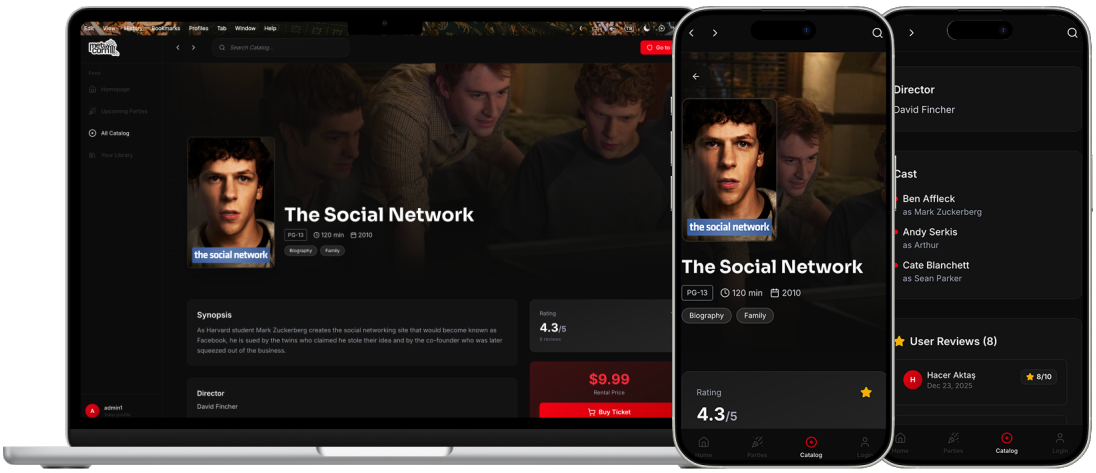

Figure 5.3: Movie Detail in Web Application of MetuCorn

This Figure 5.3 clearly states this detail page. And our sample movie page is retrieved like below, you can find the sample code how creating movie detail page works, and how we create a query.

```
import { createClient } from '@/lib/supabase/server'

export default async function MoviePage({ params }: { params: { id: string } }) {
  const supabase = await createClient()

  // Fetch movie with all related data in single query
  const { data: movie, error } = await supabase
    .from('movie')
    .select(`
      *,
      director:director_id (
        director_name,
        bio,
        photo_url
      ),
      movie_genre (
        genre:genre_id (
          name,
          description
        )
      ),
      movie_cast (
        role_name,
        cast_member:cast_id (
          cast_name,
          bio,
          photo_url
        )
      ),
      rating (
```

```
          score,
          review_text,
          client:client_id (
            name,
            avatar_url
          )
        )
      `)
      .eq('movie_id', params.id)
      .single()

    if (error || !movie) {
      return <div>Movie not found</div>
    }

    // Calculate average rating
    const avgRating = movie.rating.length > 0
      ? movie.rating.reduce((sum, r) => sum + r.score, 0) / movie.rating.length
      : 0

    return (
      <div>
        <h1>{movie.title}</h1>
        <p>Director: {movie.director.director_name}</p>
        <p>Rating: {avgRating.toFixed(1)}/10</p>
        {/* Render cast, genres, reviews */}
      </div>
    )
  }
```

5.4 | Watch Parties

Users create a watch party by selecting one of the movies they own. The form includes movie selection, time (datetime-local), maximum number of participants (2-100), and optional description fields. When the party is created, the system generates a unique 8-character party code (Math.random().toString(36).substring(2, 10).toUpperCase()). To join, the user must have an active ticket for the relevant movie; the system checks this. The host can cancel the party or update information such as time/maximum number of participants. All participants are stored in the party_participant table, and the host and other parties are listed on the party details page. When the party is created, the host is automatically added to the participant list. When the scheduled time arrives, the party becomes "active" and participants can watch the movie together.

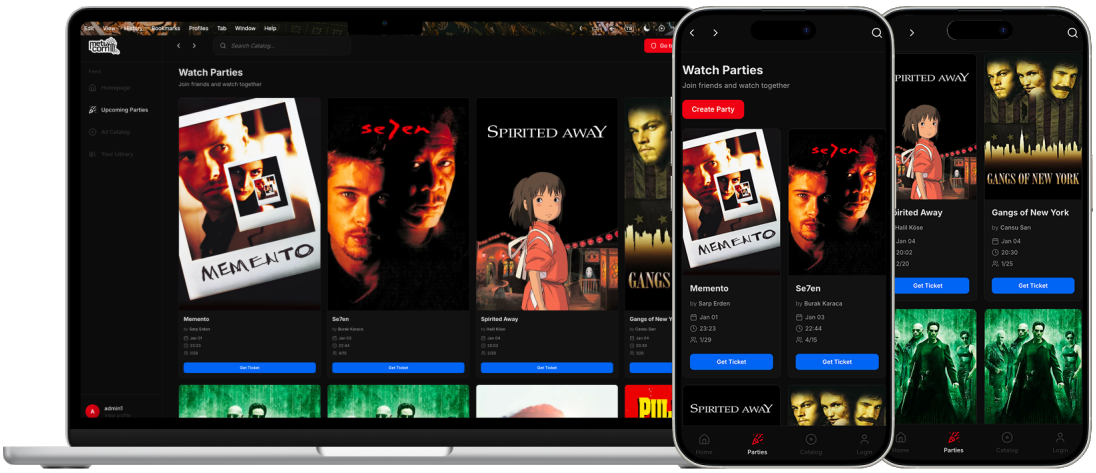

Figure 5.4: Watch Party Page in Web Application of MetuCorn

5.5 | Rating System

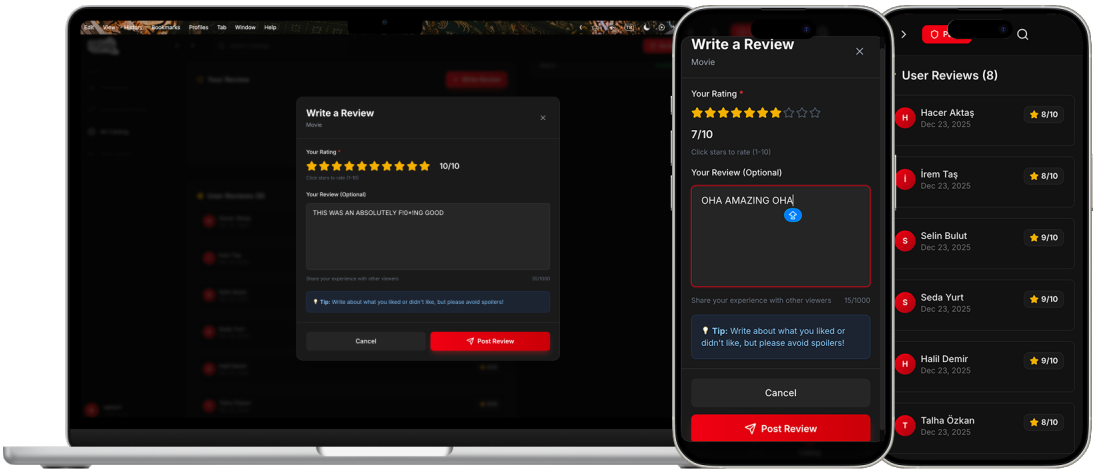

Figure 5.5: Rating System Page in Web Application of MetuCorn

In Figure 5.5 we can see our rating implementation's snapshots. Users rate movies on a scale of 1 to 10; the interface displays 10 stars, which can be clicked to assign a rating. Rating is mandatory; text comments are optional (maximum 1000 characters). Users can edit their own comments: the system checks the existing comment, updates it if necessary, or creates a new one if it doesn't exist. The average rating for each movie is calculated by dividing the total of all ratings by the number of comments and is displayed on the movie details page. The average rating is also converted to a 5-star display (the 10-point rating is divided by 2) and visualized with fractional stars. All comments are stored in the rating table, and users can only edit their own comments.

5.6 | Admin Panel

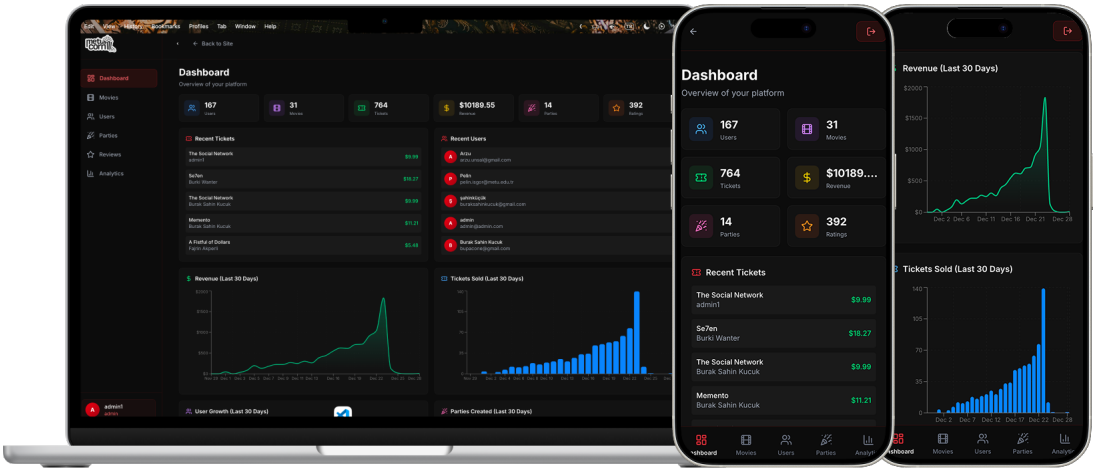

Figure 5.6: Admin System Page in Web Application of MetuCorn

As you can see in the Figure 5.6 we have very detailed admin dashboard. The admin panel provides a central control panel for platform management. The dashboard displays key metrics such as user count, active tickets, total revenue, active parties, and total comments in card format. In the film management section, admins can add, edit, and delete films; poster, director, genres, price, and status information is displayed for each film. The user management section lists all registered users; each user's name, email, phone number, number of tickets owned, and registration date are displayed. The watch party tracking page displays all watch parties; for each party, the movie, host, scheduled time, number of participants, and status (scheduled, active, completed) are tracked. The comment moderation page displays all user comments and ratings; admins can delete inappropriate content and track the average rating. The analytics

section uses the Recharts library to display graphs for the last 30 days showing revenue, tickets sold, user growth, and parties created.

As an example adding movie will work with this code in backend.

```
'use server'

import { createClient } from '@/lib/supabase/server'
import { revalidatePath } from 'next/cache'

export async function createMovie(formData: FormData) {
  const supabase = await createClient()

  const title = formData.get('title') as string
  const price = parseFloat(formData.get('price') as string)
  const status = formData.get('status') as string
  const directorId = formData.get('director_id') as string || null
  const releaseDate = formData.get('release_date') as string || null
  const posterImage = formData.get('poster_image') as string || null

  const { data: movie, error: movieError } = await supabase
    .from('movie')
    .insert({
      title,
      price,
      status,
      director_id: directorId,
      release_date: releaseDate,
      poster_image: posterImage,
    })
    .select('movie_id')
    .single()

  if (movieError || !movie) {
    return { success: false, error: movieError?.message || 'Failed to create movie' }
  }

  const movieId = movie.movie_id

  const genreIds = formData.getAll('genres') as string[]
  if (genreIds.length > 0) {
    const genreInserts = genreIds.map(genreId => ({
      movie_id: movieId,
      genre_id: genreId
    }))

    await supabase
      .from('movie_genre')
      .insert(genreInserts)
  }

  const castData = formData.get('cast') as string
  if (castData) {
    const cast = JSON.parse(castData) as Array<{ cast_id: string; role_name: string }>

    if (cast.length > 0) {
      const castInserts = cast.map(c => ({
        movie_id: movieId,
        cast_id: c.cast_id,
        role_name: c.role_name
      }))
```

```
      await supabase
        .from('movie_cast')
        .insert(castInserts)
    }
  }

  revalidatePath('/admin/movies')
  revalidatePath('/movies')

  return { success: true, movieId }
}
```

As you can see in the code it basically runs a SQL query for supabase server. To do this it first fetches data from the front-end. Form submit runs the handleSubmit function. So, form data completed then it calls createMovie. The server action (above code) adds the movie to the our database. Genre and cast relationships are added to junction tables. The cache is refreshed and the user is redirected.

6 | Conclusion

Created as part of the STAT311 Modern Database Systems course, this project effectively demonstrates the design, implementation, and deployment of a fully functional, database-based movie party platform called MetuCorn. The application was built to bridge the gap between contemporary online streaming services and traditional movie theater experiences by combining the best of both into a single, integrated solution. From a database perspective, the project successfully achieved its core objective: creating a relational database schema that supports complex real-world relationships between users, movies, tickets, payments, viewing parties, and ratings. An Entity-Relationship (ER) diagram was created to reduce redundancy, maintain data integrity, and ensure scalability. These design concepts were implemented locally using SQL. Structured and constraint-aware SQL queries provided all essential features, including ticket purchasing, viewing party participation, payment tracking, rating submission, and viewing history recording. Key analytical system features were successfully handled using advanced SQL techniques such as window functions, aggregate functions, and multiple table joins. These queries demonstrate how databases can directly aid analytical decision-making by providing informative insights into user behavior, peak viewing hours, revenue distribution by type, and customer loyalty patterns. The project also delivers an effective full-stack application beyond the database layer. A contemporary online application built using Next.js and React seamlessly integrated with a PostgreSQL database. This architecture significantly reduced development complexity while maintaining real-world industry practices. This application, utilizing the free tier of Vercel and Supabase, was thus completed.

All specified functional requirements have been completed on the web application side. Members can securely create and join viewing parties, browse movies, purchase tickets, rate and review content, and track their viewing history. Administrators are provided with a comprehensive control panel that facilitates user management, content management, and data analysis through visual reports. These features demonstrate how operational processes and analytical interfaces can be supported by a database-centric architecture.

In summary, MetuCorn is not only a useful movie viewing party platform, but also an academic example of contemporary relational database design, SQL implementation, and real-world system integration. The work demonstrates that well-thought-out data design and organized querying can provide complex user interactions while maintaining efficiency. Examples of real payment gateways, real-time chat during viewing parties, recommendation systems based on viewing history, and more analytics could be added in the future. Nevertheless, the project sufficiently meets its goals within the parameters of this course and provides a strong foundation for understanding real-world database-driven application development.

7 | Author Contributions

Fajrin Akbarli: Initiated the application concept and collaborated on the ER diagram design; contributed to the 2nd research question as well as the introduction, conclusion, and the related research-question sections of the report.

Pelin İşgör: Led the ER diagram design and documentation; authored the full ER diagram section of the report and contributed to the 3rd research question.

Ahmet Turaç: Developed the SQL architecture and implementation; authored the core SQL section of the report and implemented the 1st and 4th research questions together with their corresponding report sections.

Burak Şahin Küçük: Designed the user interface and implemented the full-stack web application; also provided support for SQL development and ER diagram design.

#END of the Report.

You can access whole application design and code below:
Best Slide of The Course By Far: Slide Completed in Figma.
Best and Only Full Stack (Still Accesible) Application of The Course: metucorn
Some Design of The User Interface and Logo: website-database Completed in Figma
GitHub Repository, Codes, Documents, Report: Github Repo

8 | References

[1] platcornweb. platcornweb on x. https://x.com/platcornweb, n.d.

[2] The Movie Database (TMDB). The movie database. https://www.themoviedb.org/, n.d.