CineTravel: Discover Movie Locations Around the World

Authors: Yuan-Hao Chen, Chia-Chun Hsiao, Sophie Huang, YuHan Huang

1. Changes in Project Direction

1.1 Added

- **Favorite Tables:** we introduced a new favorite table (and corresponding API routes) to let users mark movies they like. This lightweight "favorites" feature substituted full authentication while still enabling personalized lists.
- **Flight-location integration**: Using our GetFlightsBeforeAvg stored procedure, we now surface real flight schedules tied to each filming city. This gives users immediate insight into travel feasibility for their favorite movie locations.
- **Hotel Rating**: We added a database trigger on the hotel table that fires whenever a new rating is inserted or an existing one is updated. This trigger recomputes and writes back the hotel's Average_Score automatically, ensuring our accommodation recommendations always reflect the latest user feedback.

1.2 Removed

- **Trip Planning**: we replaced comprehensive itinerary generation with focused flight–location analytics. Hotel data remains in our location pages, but full booking flows and multi-stop trip builders were postponed.
- **User Contributions**: photo uploads and community rating tools were scoped out to concentrate on backend performance and core API stability.
- **User Accounts & Profiles**: rather than implement authentication, we opted for a simplistic "favorites" table without login, deferring true account management to future work.

2. Usefulness: Achieved vs. Failed

CineTravel delivers on its core promise: users can search movies, see authentic filming and hotel locations, and gain practical travel insights via a stored procedure that emphasizes flights departing before an airport's average time. We also added a basic favorites feature and visualized movie ratings on an interactive map, merging cinematic and hotel data with geography.

However, we postponed advanced user‑centric tools, secure authentication, community photo uploads, and in-app rating filters, to focus on stabilizing our APIs and database. As a result, social engagement and dynamic filtering remain future improvements.

3. Schema & Data Source Evolution

We simplified the movie table by removing the mainActor and genre columns, this helps us excess attributes and streamlined joins with location and flight data. We also introduced a new favorite table (Id_movies, title) to persist user bookmarks without implementing authentication. Additionally, we implemented a MySQL trigger on our hotel‑rating inputs so that whenever a new rating is inserted (or an existing one updated). These adjustments focused the schema on high-value relationships and enabled efficient queries for both movie lookup and personalized favorites.

4. ER Diagram & Table Implementation Changes

4.1 Changes to our ER diagram

1. Replaced the Accommodation table with a new Hotel table.
   In the original design, Accommodation had fields like hotel name, hotel price, hotel review, and hotel location. In the new hotel table, we introduced new attributes:
   Hotel_country and City fields to separate out the country and city explicitly, making geographic queries easier (for example, filtering hotels by city or country).
   Average_Score to store the hotel's average review score numerically, instead of vague text-based reviews.
2. Add Airport table
   Instead of mixing airport details into other tables, we created a dedicated Airport table to store important information like:
   - Code (primary key),
   - Name (name of the airport),
   - City and Continent (where the airport is located)

   Having a separate Airport table makes it easier to update or expand airport information in the future (for example, adding fields like country, timezone, or airport size) without changing other parts of the database.
   In addition, since the Airport table has a City attribute, it allows us to join airport information with the movie_location table, which also stores city-related information.
   This connection makes it possible to create new features, such as finding flights to cities where movies were filmed, enhancing the travel and movie exploration experience.
3. Remove User table
   Since the primary focus of the system is browsing movie locations and flights, the User table was unnecessary. Removing it simplifies the database structure, reduces complexity, and allows for faster development without impacting the core functionality.
4. Add favorite table
   We added a new table called Favorite to our ER diagram to support the feature where users can like and save their favorite movies.
   The Favorite table has two fields:
   - id_movies, which stores the movie's ID
   - title, which stores the movie's title

   This table gives users a way to keep track of the movies they like without needing to create an account or log in.
5. Modify Movie table
   We added important attributes on the movie table such as vote_average (rating score) and release date (year, month, day) to better support sorting, filtering, and displaying movies based on their popularity or release time.
6. Combine Movie_Location and Location table:
   Originally, the ER diagram separated the information like: Location stored general geographic information like country, state, city, and address. Movie_Location linked specific movies to locations using a Location_id and Movie_id. However, we found that separating them added unnecessary complexity without strong benefits for our goals. Since our project focuses on browsing movies by filming locations globally, it made more sense to merge these two concepts into one unified table.

In the new movie_location table:

- We directly store Continent, Country, and Locations along with the Movie title and Director name.
- We use composite primary keys (Continent, Country, Movie) to ensure each record is unique and easy to reference.

4.2 Table implementations

- Advanced queries 1:
  We implemented the Explore Movies by City page using a query that joins movie_location and airport tables.
  This query matches movie filming locations with airport cities, using a case-insensitive join (LOWER() function).
  We then group airports by city and count the number of airports (flight connections), helping users find movies filmed in cities with available flights.
- Advanced queries 2:
  On the Location page, we show average movie ratings grouped by filming country.
  This query joins the movie table with movie_location, aggregates the data with GROUP BY, and sorts results.
- Stored procedure:
  We implemented a Stored Procedure called GetFlightsBeforeAvg on Flights Before Avg Depart Time page. It allows users to find flights from a given origin that depart earlier than the average scheduled departure time for that airport.The procedure uses a subquery to calculate the average departure time dynamically and filters flights accordingly.
- Keyword Search:
  We implemented keyword search through a dropdown menu that allows users to select a release year. When a year is selected, the frontend sends a request with the chosen year as a query parameter to the backend. The backend then executes a SQL query that filters movies by the selected year.
- CRUD:
  We implemented the Create, Read, and Delete operations from the CRUD model through the **"Like" button** functionality. When a user clicks the Like button on a movie, a **Create** operation is triggered, storing the movie into the user's Favorite List. The platform also performs a **Read** operation by fetching and displaying all the movies the user has previously liked, allowing users to easily view and manage their favorite films. If a user changes their mind, they can click to Unlike a movie, which triggers a **Delete** operation, removing the movie from the Favorite List. Through these interactions, users can personalize their experience by keeping track of the movies they are interested in for future trip planning.

  Users can update a specific hotel’s rating according to their personal experience, which will satisfy the update requirement. And this update will trigger a storage for changing records.
- Trigger**:**
  When a hotel rating is updated, our sophisticated trigger system automatically captures and records both the previous and new ratings, creating a comprehensive historical timeline of quality fluctuations. This dynamic tracking system serves two critical purposes: it enables prospective guests to analyze recent trends in service quality before making booking decisions, while simultaneously providing hotel management with actionable intelligence about performance patterns. Through this transparent rating history, travelers gain deeper insights into a property's consistency and trajectory, while hoteliers can identify specific turning points in guest satisfaction, allowing them to implement targeted policy adjustments and service improvements that directly address emerging concerns or capitalize on successful initiatives.

Our final design is more suitable because it gives us a cleaner and simpler structure. By reducing the number of tables and making each one more meaningful, we made querying much faster and easier for us to manage.
We also achieved better normalization by properly separating flight and airport data, as well as hotel and location information.This helps us keep the data consistent and organized as our project grows.It also sets us up for easier expansion in the future.
With this way, it will be straightforward to add features like filtering movies by rating, finding nearby hotels, or recommending flights based on filming locations.

5. Functionalities: Added or Removed

5.1 Added

- **User Favorite Movies:**

  We added a favorite feature that lets users mark movies they like.
  Instead of building a full authentication system, we wanted to still offer users a personalized experience in a lightweight, easy-to-maintain way. This keeps the platform simple while still giving users a way to "save" their interests.

- **Flight-Location Integration:**

  We introduced flight data into the browsing experience using our GetFlightsBeforeAvg stored procedure.
  Adding flight schedules tied to movie locations makes the experience more practical and travel-focused, directly supporting our goal of connecting cinematic exploration with real-world travel planning.

- **Hotel Rating:**

  We implemented a database trigger on the hotel table that fires whenever a new rating is inserted or an existing one is updated. This trigger recalculates and writes back the hotel's Average_Score automatically, ensuring our accommodation recommendations always reflect the latest user feedback.

5.2 Removed

- **User Login System:**
  We removed the full user login, registration, and authentication system.
  Building secure login infrastructure would have significantly increased complexity. Since our main goal is movie and location exploration, not managing accounts, we replaced it with the simpler "favorites" feature to stay focused.

- **Trip Planning Functionality:**
  We removed full itinerary generation and trip booking flows.
  These features would have required complex multi-city scheduling and real-time booking APIs. Instead, we focused on providing flight-location insights, which still help users without the need for full travel booking infrastructure.

- **User Contributions (Photo Uploads, Community Ratings):**
  We postponed features like user-uploaded photos and community-based movie ratings. Implementing social engagement would have added significant backend and moderation challenges. We prioritized API stability, data accuracy, and core functionality first to ensure a solid foundation for future expansion.

5.3 Current Functionalities

- **Movie Information:**
  Users can browse movies by release year, view movie titles, and like their favorites directly from the homepage.
- **Favorite Movies Feature:**
  We introduced a favorite table along with corresponding API routes that allow full Create, Read, and Delete operations:
  Users can like (create) a favorite movie,View (read) all their favorite movies, and remove (delete) movies from their favorites list.
- **Favorite Movies Page:**
  Users can view and manage their list of favorite movies, even if they have not logged in.
- **Flight-Location Integration:**
  Using the GetFlightsBeforeAvg stored procedure, we show real flight schedules tied to each filming city.
- **Interactive Map:**
  We implemented an interactive map that displays filming locations, top movies, hotels, and average movie ratings by country.
- **Explore Movies by City with Flight Count:**
  Users can select a city, view movies filmed there, and see the number of flights connected to that city.

6. Complementarity of Advanced Database Programs

- **Advanced Query Integration:**
  We wrote SQL queries that join movie location data with real airport data, counting the number of available flights for each city. Allowing the frontend to dynamically display not only movies but also flight availability tied to the filming locations selected by users.

  We also utilized sophisticated SQL operations, including aggregation (GROUP BY) and multi-table joins, to calculate accurate average movie ratings across different filming countries. This enables users to quickly identify and explore highly-rated filming destinations on the interactive map.Complementing these advanced queries, we integrate the calculated data directly into the Google Maps API, dynamically displaying filming locations on an interactive map. Each location visually presents its average movie rating and highlights the top three highest-rated movies filmed there.

- **Stored Procedure Implementation:**
  We created a stored procedure, GetFlightsBeforeAvg, that retrieves flights departing earlier than the average scheduled time from a selected airport.
  This procedure uses input parameters, subqueries, and aggregation functions to perform advanced logic inside the database, reducing frontend complexity and improving query performance.

- **Constraints**

Throughout our database design, we enforced primary keys, and attribute-level constraints such as NOT NULL and appropriate data types.
We also used composite primary keys where needed to maintain data integrity, for example in the movie-location relationships.

- **Trigger**
  We implemented database triggers to automate critical processes and maintain data consistency throughout our travel and movie database system. Specifically, we created the hotel_rating_update_trigger which activates automatically after any hotel rating update. This trigger performs two essential functions simultaneously:
  It records all rating modifications in the hotel_rating_history table, preserving valuable historical data including the previous rating, new rating, affected country, and timestamp. This provides a complete audit trail of all rating changes for data analysis and accountability purposes.

7. Technical Challenges & Lessons Learned

YuHan Huang: The main technical challenge for the location part will be setting up the interaction of the html button and the corresponding reaction after getting API results. Another part of the challenge will be researching how to use google map API, including what is the ideal input and output of the API and how can I use it.

Yuan-Hao Chen: One of the technical challenges we faced was ensuring proper synchronization between the frontend and backend, especially when handling actions like adding and removing favorite movies. We resolved this by using Observables to manage asynchronous data fetching, ensuring the UI updates in real-time.

Chia-Chun Hsiao: One technical challenge we faced was keeping the backend APIs consistent with what the frontend expected. We built an API for users to search movies by city, but quickly realized city names in the database and user input didn't always match due to casing differences (like "Chicago" vs. "chicago"). To fix this, we updated the backend SQL queries to use LOWER() and made sure the frontend sent city names in lowercase. This simple change avoided a lot of empty search results. It taught us that small issues like casing can cause bigger integration problems, and it's important to agree early on a consistent data format between frontend and backend.
Sophie Huang: In our final GetFlightsBeforeAvg stored procedure, we implemented a nested subquery that calculates the average departure time per origin using STR_TO_DATE and AVG, then filters rows in the outer query where individual departure times fall before this average. I removed the LIMIT clause to return all matching flights for pagination. A key challenge was handling time strings with STR_TO_DATE, which prevented index usage; we addressed this by adding a functional index on the converted time expression so the optimizer could use it. Additionally, calling the procedure returned multiple result sets, requiring extraction of the first array to access the records.

8. Other Changes vs. Original Proposal

Beyond the core focus on movie–flight integration and favorites, we have listed our major project directions changes at Section 1, 2, and 5.

9. Future Work

In future iterations of CineTravel, we plan to implement full user accounts with secure authentication to enable personalized experiences and protect user data. We will also integrate interactive maps that display pin-drops for filming locations alongside the nearest airports, giving users a visual itinerary builder. Finally, we are going to develop an enhanced recommendation engine that combines user favorites, location ratings, and travel patterns to generate customized, multi-stop itineraries tailored to individual preferences.

10. Division of Labor & Teamwork

YuHan Huang: Location page, Google Map API, web crawling for database
Yuan-Hao Chen: Homepage, Liked button and backend API connection
Chia-Chun Hsiao: Explore Movies by City page and backend API connection
Sophie Huang: Stored Procedure, Flight Query page and backend API connection