MovieDB System

CMPE321 - Introduction to Database Systems

Project 3

Spring 2023

Koray Tekin - 2018400213

Leyla Yayladere - 2018400216

Table of Contents

Design Analysis

During the analysis of our design, we discovered that we had made efforts to fulfill the constraints outlined in Project 1 through our ER design. However, in this project, we have realized that certain constraints can now be effectively enforced using the CHECK/TRIGGER construct or implemented through backend functions. For example, in order to satisfy the review constraint, we utilized 3 different aggregation and merged tables through relations, resulting in a degree of redundancy within our design. Although our approach introduced complexities, we chose to keep the design since redundancy is not a problem for the scope of this project. We think that the complexity of the design enhances the overall reliability of the database. Additionally, we will detect possible problems associated with redundancy such as insert/delete/update anomalies using FDs.

FDs and Normal Forms

List of all the non-trivial functional dependencies for each table is as follows:

1. Table: db_manager
    - username → username, password
      Analysis: The table satisfies BCNF since the determinant (username) is a candidate key, uniquely determining all other attributes.

2. Table: audience
    - username → name, surname, password, username
      Analysis: The table satisfies BCNF since the determinant (username) is a candidate key, uniquely determining all other attributes.

3. Table: director
    - username → name, surname, password, username, nation
      Analysis: The table satisfies BCNF since the determinant (username) is a candidate key, uniquely determining all other attributes.

4. Table: experienced_director
    - username → name, surname, password, username, nation
      Analysis: The table satisfies BCNF since the determinant (username) is a candidate key, uniquely determining all other attributes.

5. Table: rating_platform
    - platform_id → platform_name
    - platform_name → platform_id
      Analysis: The table satisfies BCNF as both FDs are superkey dependencies, with the determinant determining all other attributes.

6. Table: movie
    - movie_id → movie_id, movie_name, average_rating, reviewer_num
      Analysis: The table satisfies BCNF since the determinant (movie_id) is a candidate key, uniquely determining all other attributes.

7. Table: genre
    - genre_id → genre_name
    - genre_name → genre_id
      Analysis: The table satisfies BCNF as both FDs are superkey dependencies, with the determinant determining all other attributes.

8. Table: movie_session
    - date, time_slot → date, time_slot (trivial but only FD of this table)

9. Table: theater
    - theater_id → theater_id, theater_name, theater_capacity, theater_district
      Analysis: The table satisfies BCNF since the determinant (theater_id) is a candidate key, uniquely determining all other attributes.

10. Table: in_host
    - date, time_slot, theater_id → date, time_slot, theater_id (trivial but only FD of this table)

11. Table: in_session
    - session_id → movie_id, session_id, date, time_slot, theater_id
    - date, time_slot, theater_id → session_id
    - date, time_slot, theater_id → movie_id, session_id, date, time_slot, theater_id
      Analysis: The table satisfies BCNF as all FDs are superkey dependencies, with the determinant determining all other attributes.

12. Table: contract
    - username → username, platform_id
      Analysis: The table satisfies BCNF since the determinant (username) is a candidate key, uniquely determining all other attributes.

13. Table: directed_by
    - movie_id → movie_id, username, platform_id
    - username → platform_id
      Analysis: The table **does not satisfy BCNF** because the second FD has a determinant (username) that does not functionally determine all other attributes. Additionally, it **does not satisfy 3NF** because the dependent part (platform_id) of the second FD is not a part of superkey.
      Decomposition into BCNF: We used the lossless join decomposition technique mentioned in the slides to decompose into BCNF. New tables are (movie_id, username) and (username, platform_id). Luckily, we already have (username, platform_id) as a contract relation, so we can revise this table by removing platform_id attribute. Since each director can have at most one platform, it's naturally lossless join decomposition. Also all dependencies are preserved!

14. which_genre(movie_id : integer, genre_name : string)
    - movie_id, genre_name → movie_id, genre_name (trivial but only FD of this table)

15. preceded(child_movie_id : integer, predecessor_movie_id : integer)
    - child_movie_id, predecessor_movie_id → child_movie_id, predecessor_movie_id (trivial but only FD of this table)

16. subscribe(audience_username : string, platform_id : integer)
    - audience_username, platform_id → audience_username, platform_id
      (trivial but only FD of this table)

17. ticket(ticket_no : string, purchaser : string, session_id : integer, movie_id : integer)
    - ticket_no → purchaser, session_id, movie_id
    - session_id → movie_id
      Analysis: The table **does not satisfy BCNF** because the second FD has a determinant (username) that does not functionally determine all other attributes. Additionally, it **does not satisfy 3NF** because the dependent part (platform_id) of the second FD is not a part of superkey.
      Despite these normalization issues, we have chosen to leave this table as it is. The decision to retain the current design is based on the requirement of fulfilling the reviewing process, since table includes all the necessary attributes to meet the requirements of the reviewing process

18. review(rating : real, ticket_no : string, subscribed_audience : string, session_id :integer, movie_id : integer, reviewer_platform_id : integer)
    - subscribed_audience, move_id → rating, subscribed_audience, session_id, move_id, reviewer_platform_id
    - ticket_no → subscribed_audience, session_id, movie_id
      Analysis: The table **does not satisfy BCNF** because the second FD has a determinant (username) that does not functionally determine all other attributes. Additionally, it **does not satisfy 3NF** because the dependent part (platform_id) of the second FD is not a part of superkey.
      We realized that keeping session_id attribute in this relation is unnecessary since we can reach it by ticket table using ticket_no. By removing this attribute, the second FD changes as follows
    - ticket_no →subscribed_audience, movie_id
      Analysis: The table satisfies BCNF as both FDs are superkey dependencies, with the determinant determining all other attributes.

Schema Refinement Step

Change in our createTable.sql

→ in order to have directed_by relation in BCNF
→ **removed platform_id attribute from directed_by**
→ for audience deletion ang
→ **added reviewer_num attribute into movie table**
To preserve the rating effect of an audience even after their deletion, we added this attribute into movie table. When an audience is deleted, all their personal data, including subscriptions and tickets, are also removed. Since our review table references ticket table, reviews associated with the deleted audience must also be deleted. As a result, if we want to retain the ratings of the deleted audience, we cannot use review table. This attribute makes it possible to calculate average_rating as average_rating * reviewer_num + new.review.
→ in order to have directed_by relation in BCNF
→ **removed session_id attribute from review**
→ for average_rating of movies constraints

→ **created update_avg_rating trigger**

The update_avg_rating trigger was created to update the average_rating and reviewer_num attributes in the movie table whenever a new review is inserted into the review table. This trigger recalculates the average_rating based on the existing values and the new review, and updates the reviewer_num accordingly. This ensures that the average_rating reflects the cumulative ratings from all reviewers and keeps the constraints on the average rating of movies intact.

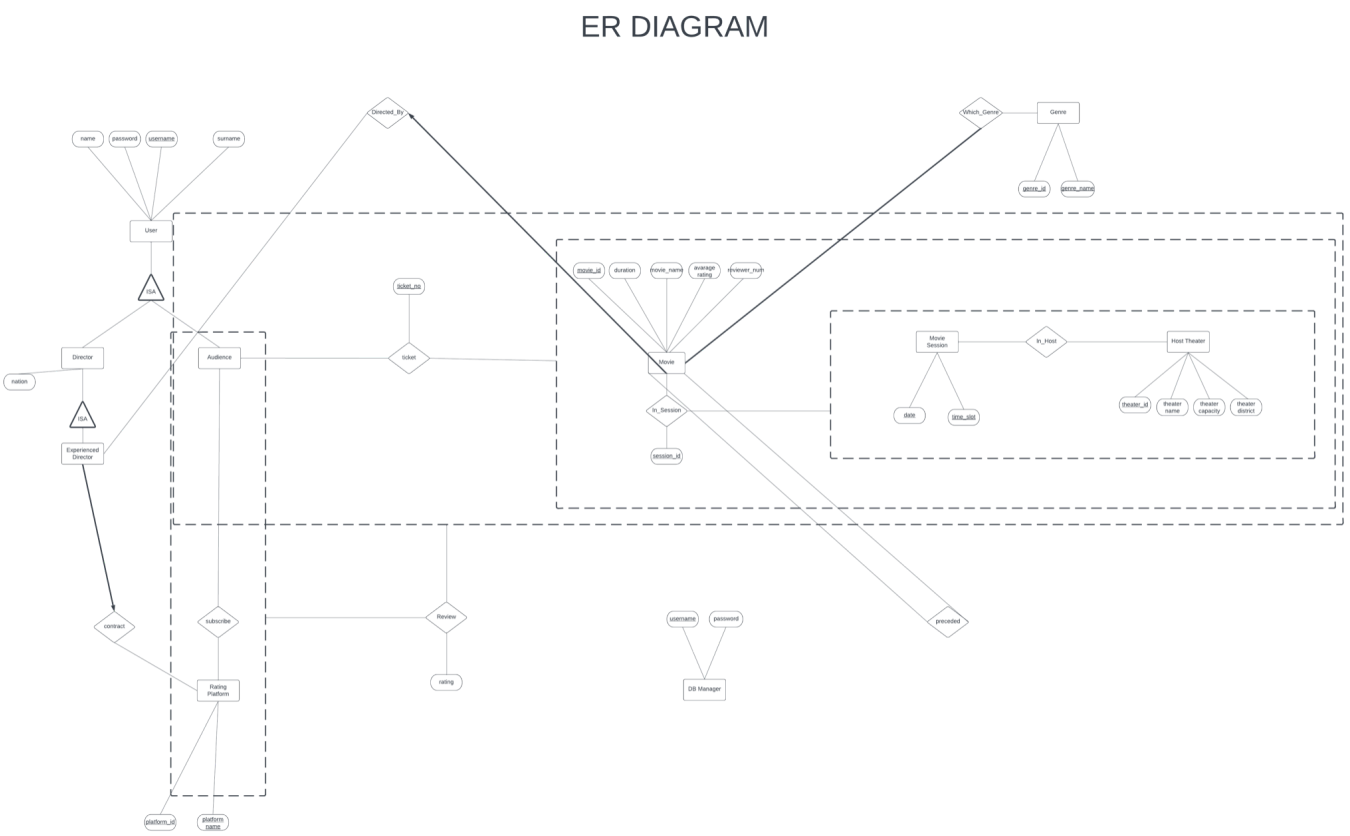
```
CREATE TRIGGER movie_db.update_avg_rating
AFTER INSERT ON movie_db.review
FOR EACH ROW
BEGIN
  DECLARE existing_rating FLOAT;
  DECLARE existing_reviewers INT;

  SELECT average_rating, reviewer_num INTO existing_rating, existing_reviewers
  FROM movie_db.movie
  WHERE movie_id = NEW.movie_id;

  UPDATE movie_db.movie
  SET average_rating = (existing_rating * existing_reviewers + NEW.rating) / (existing_reviewers + 1),
    reviewer_num = existing_reviewers + 1
  WHERE movie_id = NEW.movie_id;
END
```

dropTable.sql remains the same.

Revised ER Diagram



You can reach the ER diagram of our design of MovieDB on Lucidchart via this link, if you wish to see it in more detail.

Constraints Handled

Firstly we want to mention that, during the first project in order to satisfy as much requirements as possible; we have benefitted from various foreign key / references relationships. During this project we have implemented some of the remaining constraints via back-end application, some of them via modified tables and some via the usage of “triggers” as requested in the description the SQL related refinements are mentioned in the related section, in this part we will touch upon the constraints handled via back-end:

- Session/Slot relation: The directors have the ability to create a movie session via the related form located in the director page, during this creation they have to specify theater id, date and time slot. When handling movie session addition request, after initial validations of the inputs of the form we go and check the related table for existing sessions and whether another movie either starts or continues in the specified theater at the specified time. If this is the case we do not allow the director to add this session of his/her movie.

- Key relations: In order to not experience disquietive errors on the database management side we have also created an additional layer of validation/verification and for most of the database accesses, we initially checked the query input parameters. In order to exemplify the process assume that a manager wants to create a director, he/she has a form in front and there are several fields to fill; it is possible for the manager to enter an invalid data to either of those fields or get confused and try to add a review platform contract for an already existing director via the creation form. In any of those cases our validation layer disables the director to conduct a misevaluated operation. This is achieved via several “selections” from the existing database relation instances.

After the check for the violation of these constraints, we inform the user about the unsuccessful operation via pop-up messages on the related page. This process is mentioned at the end of next section.

Explanation of Implementation

First of all due to this project’s requirements we were expected to come up with a front-end design, back-end design and a database connection from the back-end design which would enable us to benefit from the revised tables that we have created in the first project. For these reasons, we began with a relatively simple full-stack Django application design. We will be inspecting the implementation of our

application under two fundamental titles, namely “Front-end” and the “Back-end/Database”.

Front-end:

For the front-end of our application we began with creating 6 main pages, namely; “manager-login”, “director-login”, “audience-login” and following these pages respectively redirecting to “manager”, “director” and “audience” pages. The first 3 pages require the users to enter their credentials to log-in to the system and access to their pages. The verification of those credentials are handled at the back-end and if the user correctly enters the credentials, she/he is able to log-in and use the other functionalities of the application. The manager credentials are added to the database schema as specified in the description tables. The directors and audience members need to be added by the manager, then they are also able to log-in via their respective pages. These 3 second-layer pages contain several forms, the input of those forms are captured and used as parameters for the corresponding requests when the submit button is activated.

- The forms that manager page contains are: “Add New Audience”, “Add New Director”, “Update Director Platform ID”, “View All Directors”, “View Director's Movies”, “View Audience Ratings”, “Remove Audience”, “View Movie Rating”.

- The forms that director page contains are: “List Available Theaters”, “Add Theater”, “Add Movie Session”, “Add Genre to Movie”, “Add Predecessor Movie”, “List My Movies”, “Update Movie Name”, “List The Audience”.

- The forms that audience page contains are: “List All Movies”, “Purchase Movie Ticket”, “List My Tickets”, “Subscribe to Rating Platform”, “Rate a Movie”.

Back-end / Database:

To begin with for the back-end, database connection we have used the Django’s Database connection module and specified a connection to “Mysql” servers that are running on the local computer. Via this set-up we were able to handle the query calls to database at the endpoints of back-end when triggered via the front-end functionalities.

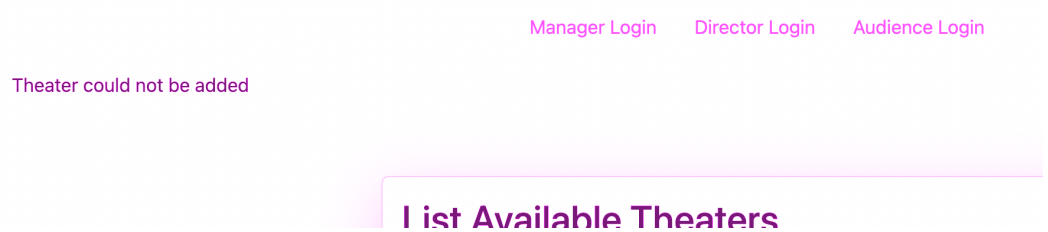
As we have done in the front-end we have gone page by page through implementation, for each form we have created a corresponding back-end function that receives the parameters of the function and after conducting the necessary processing (we have checked for; empty form submissions, invalid value entries for any of the fields, etc.) and the corresponding table checks, the functions conduct an insertion/update if required and return the values that are supposed to be depicted on the front-end or a success message if the insertion/update is completed successfully.

One of the important architectural approaches can be considered as the “User Identification”. This was required since, when a director or audience member logs into the system and tries to view their movies/tickets, this request form does not contain a parameter and the responsibility of acquiring the movies/tickets for the corresponding user belongs to us. We decided to hold the credentials of the last logged in user for each session at local storage and used this information while the function calls to back-end.

Invalid Input Handling

As mentioned in the other sections in case of invalid input (already existing entry,addition to already full slot, empty username, etc.) on the forms at the pages; manager, director, audience, we handle the unsuccessful request via not modifying the relations’ states and informing the user via related messages as “User can not be added” which pop up at the top of the page. In case of requests that require the depiction of a list if the specified parameters are invalid or the requested list is empty, even though the user is redirected to the same page the related list field can be observed as emty.

Example Error Message(In case of already existing theater_id usage):



README.md

You can find it in the main directory of the application package.

Conclusion

In conclusion, we have successfully addressed all the constraints and requirements outlined in this project. We have incorporated the necessary constraints into our database design and implemented additional functionalities through our backend functions. Moreover, we have enhanced the user interface (UI) by adding extra features such as "Add Theater" and "Add Genre to Movie" for logged-in directors, as well as "Rate a Movie" for logged-in audiences. These additional functionalities were not mandatory but have been included to simulate a wide range of scenarios within UI. Our UI is capable of handling all the necessary operations and interactions required for this Movie System.