The Chinese University of Hong Kong, Shenzhen

CSC 3170

Database System

Group 26 Report: Rotten Potatoes

Authors:	*Student Number:*
郑时飞	119010465
朱伯源	119010485
李易	119010156
施天昊	120090472
汪明杰	119010300

Thursday 12th May, 2022

Contents

1 Introduction

As an art and commercial work, movies have become pursuits of many people and generated tremendous value. Currently, there are two types of online movie platforms. On the one hand, platforms including Rotten Tomatoes and Douban gives people rich information regarding movies. On the other hand, sites like Netflix offers precise, personalized recommendation to users. Such a recommendation strategy allows users to discover more films they enjoy and attracts more users to the platform.

However, currently, no site combines the strengths of the mentioned movie sites. Besides, existing popular movie websites are full of quarrels and controversies from the perspective of the community environment. The movie recommendation sare influenced deeply by the advertising, not the quality of films. Therefore, to satisfy the real requirements of movie lovers, we build a movie database called **Rotten Potatoes**, based on which, users can talk about their reviews on the movie freely. It also offers plenty of searching functions and personalized recommendations from the platform.

We use python packages ***Requests***, ***BeautifulSoup*** to obtain the required data from IMDB, store it in a MySQL database. Then we build our searching functions and personalized moive recommendations on those data. We also build a user-friendly web using ***amis*** as the front end and ***express*** as the back end.

2 Design

In this section, we focus on the design of Entity-Relationship Model, the reduction from ER diagram into relational schemas, constraint and index.

2.1 Entity-Relationship Model

As shown in the ER digram below, there are 6 entities and 5 relationships.

Entity "movies" To store id, name, cover url, introduction, release year and genres of a movie, where genres is a multivariate attribute. Identified by id.

Entity "directors" To store id, name, photo url, introduction and birth date of a director. This entity has a one-to-many relationship "direct" with the entity "movies", which means a director can directs multiple movies and a movie can be directed by only one director in our assumption. Identified by id.

Entity "actors" To store id, name, photo url, introduction and birth date of an actor. This entity has a many-to-many relationship with the entity "movies", which means an actor can act in multiple movies and a movie can be acted by multiple actors in our assumption. Identified by id.

Entity "users" To store id, name, avatar url, password of a user. This entity has a many-to-many relationship with the entity "movies", which means a user can comment on multiple movies and a movie can be commented by multiple users in our assumption. Identified by id.

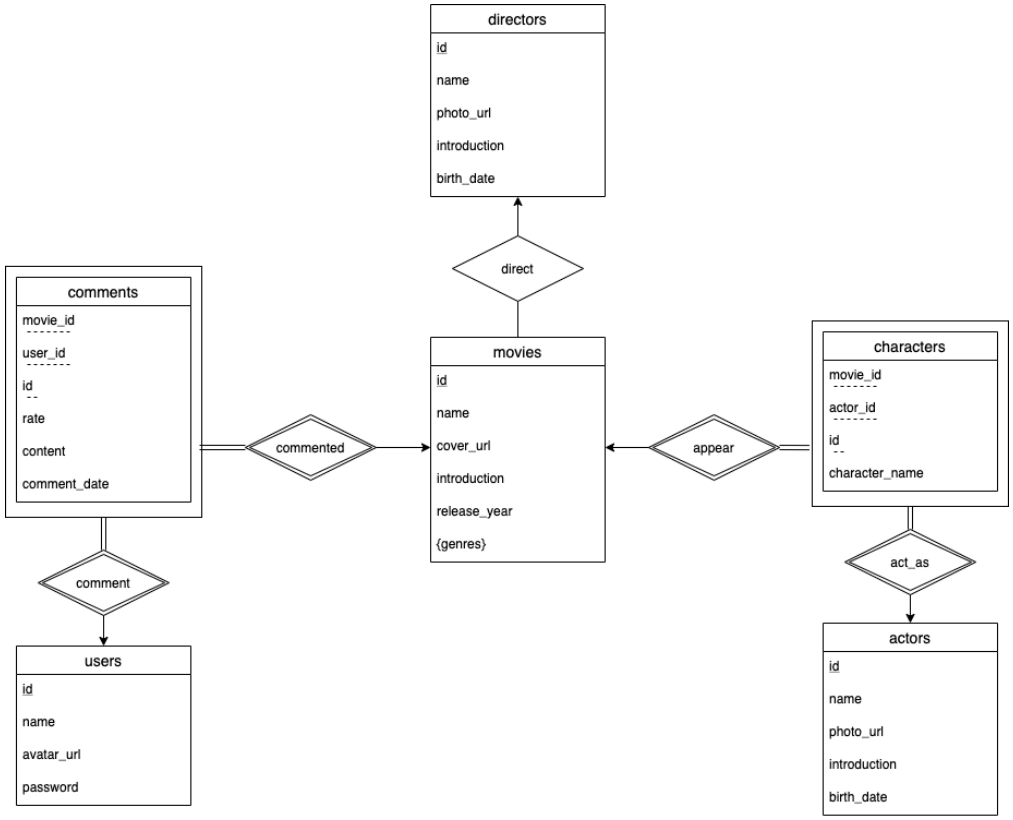


Figure 1: ER Diagram

Entity "characters" To store movie id (of the movie where this character appears), actor id (of the actor who acts as this character), id, character name of a character. This entity is a weak entity identified by entity "movies" through relationship "appear" and entity "actors" through relationship "act as", and also by its own id, which means a character can be acted by exactly one actor and appear in exactly one movie in our assumption (we treat characters of the same name appearing in multiple movies or acted by multiple actors as multiple different characters for simplicity). Note that since this entity is also identified by its own id, it is allowed that an actor acts as multiple characters in the same movie.

Entity "comments" To store movie id (of the movie commented by this comment), user id (of the user who makes this comment), id, rate (from 0 to 10), content and comment date of a comment. This entity is a weak entity identified by entity "movies" and entity "users", and also by its own id, which means a comment is on exactly one movie and is made by exactly one user in our assumption. Note that since this entity is also identified by its own id, it is allowed that a user makes multiple comments on the same movie.

2.2 Relational Schema

As shown in the relational schema digram below, there are 7 schemas.

The following reductions are made:

- The attribute "genres" in entity "movies" is reduced to schema "genres" with attributes "genres_name" and "movie_id" as a foreign key, both of which forms a primary key to make

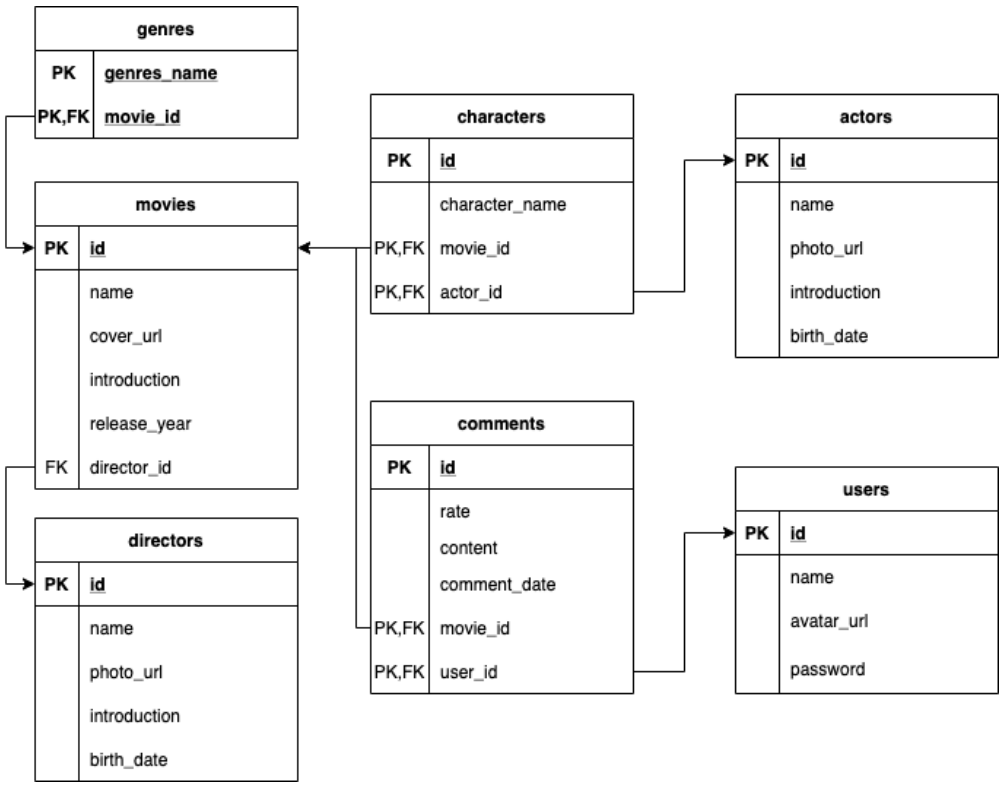


Figure 2: Relational Schema Diagram

sure no redundant genres of a movie.

- The relationship "direct" between entity "movies" and "directors" is reduced to attribute "director_id"as a foreign key in schema "movies" so that a movie is directed by exactly one director.
- All attributes identifying the entity are reduced to primary keys.
- "movie_id", "actor_id" of entity "characters", "movie_id", "user_id" of entity "comments" are reduced to foreign keys in their schemas referencing their corresponding identifying strong entities.

2.3 Constraint

3 types of constraints are further added:

Not Null Constraints

- All "id" attributes as they are primary key and thus automatically becoming not null.
- "name" attribute of schema "movies".
- "name" and "director_id" attributes of schema "directors", as a movie is directed by exactly one director.
- "name" attribute of schema "actors".

- "name" and "password" attributes of schema "users".
- "actor_id", "movie_id" and "character_name" attributes of schema "characters", as it is a weak entity of schemas "actors" and "movies".
- "user_id", "movie_id", "rate", "content" and "comment_date" attributes of schema "comments", as it is a weak entity of schemas "users" and "movies".
- "genres_name", "movie_id" attributes of schema "genres", as it is a multivariate attribute of schema "movies".

Unique Constraint A unique constraint is added to "name" attribute of schema "users" as by our assumption there should be no repeating user names.

Check Constraint A check constraint is added to "rate" attribute of schema "comments" to make sure the rate is from 0 to 10.

2.4 Index

The following attributes are indexed to make search faster:

- "name" and "release_year" attributes of schema "movies".
- "name" and "birth_date" attributes of schema "directors".
- "name" and "birth_date" attributes of schema "actors".
- "name" attribute of schema "users".

3 Implementation

3.1 Frontend

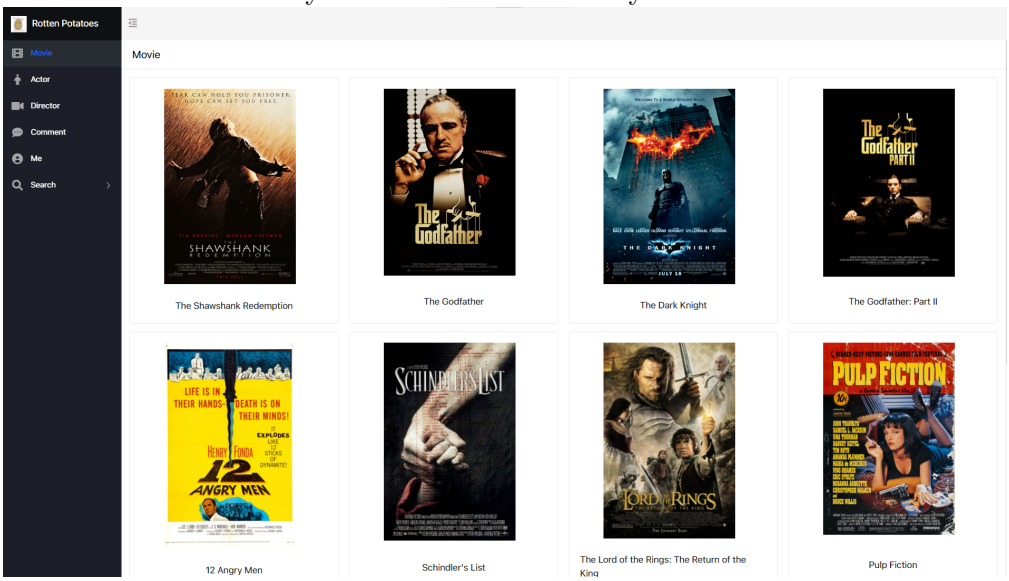
The frontend of our project is constructed using AMIS, a low-code front end framework. It can generate a website using json configurations, which is suitable for developing a light-weight and agile application like this project.

When users access the webpage, they will be directed to the Login page, where registered user can input user name and password to log in. We also added support for new user registration. After login, users can see the main structure of our website, which features a navigation pannel on the left side and the content on the right. We constructed the following pages:

- Movie: a list of all the movies with their name and posters. Each movie is a "Card" widget linking to the movie detail page.
- Actor: a list of all the actor/actresses, also implemented using the "Card" widget, containing links to the detailed information.
- Director: a list of all the directors implemented similarly as the **Actor** page
- Comment: this page contains the latest comments that users release.
- Me: a portal for users to edit their information. Including updating avatar, name, password. This page also contains a list of movies recommended to the user.

- Search: to search for movies/actors/user, we implemented three different pages. The details will be discussed in **Sample Queries** section.

The figure below shows the movie page of our website. Since our application is user-oriented, the UI of our website is very concise and user-friendly.



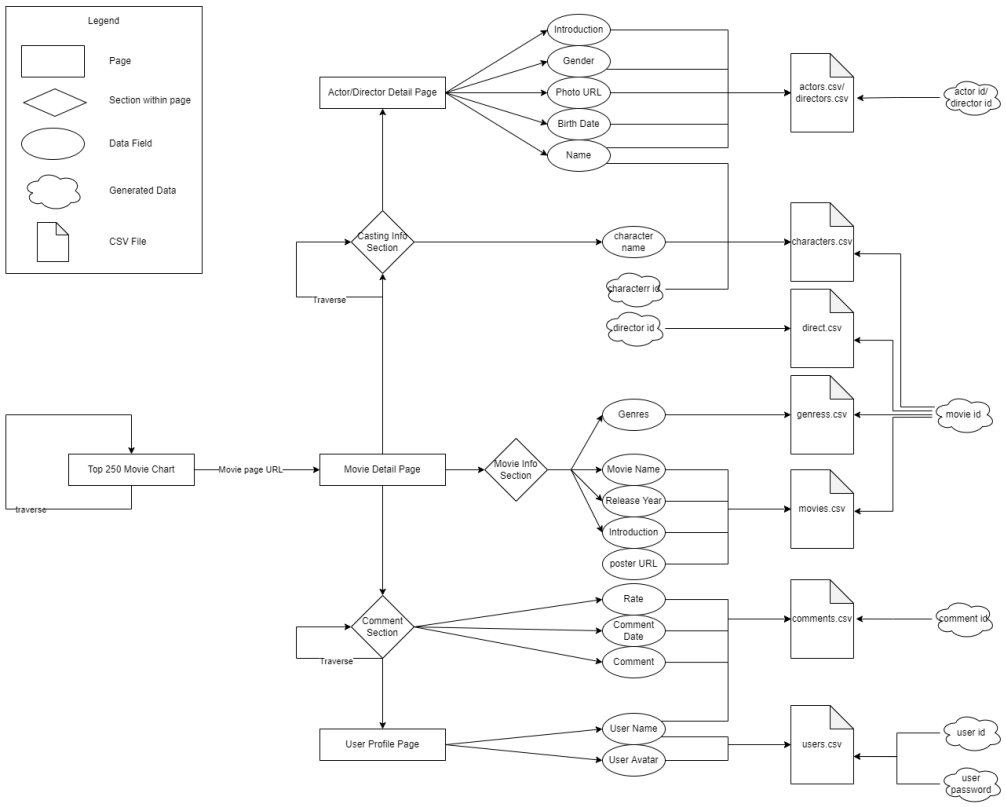
3.2 Backend

Backend of this project is implemented through the express package of javascript. During the initialization, the 'init.js' will read data from CSV files and create queries to batch-insert records into corresponding tables.

3.3 Web Crawler

3.3.1 Basic Workflow

To populate our database with real-world data, we wrote a web crawler to scrap information from IMDB's Top 250 Movie Chart. The crawler is written in Python. It uses the requests library to send https requests. The desired fields are acquired through parsing the page using Beautiful Soup 4 library. The detailed process is as follows:



1. Access the Top 250 Chart, where all the URLs to the detailed movie pages are located
2. Traverse through the chart. For every movie of the chart:
   (a) Access the detailed movie page, where information regarding the movie can be found.
   (b) From the detailed movie page, we access the casting information.
   For the director:
   - For directors yet to be recorded, access the detailed director page, where information regarding the director can be found.

   For the actors/actresses who casted in this movie:
   - Access the detailed actor/actress page and find detailed information.

   (c) The comment section locates at the bottom of the movie page. We collect the rating, comments from this section.
   (d) For each comment, we acquire the user information by accessing their homepage.

3.3.2 Encountered Problems and Solutions

During the scrapping process, we encountered some issues.

- The same director or actor/actress can participate in different movies. Therefore, we keep a mapping relationship and check wheter we have record the same person's information. This step is done through hashing the person's name, which takes O(1) complexity.
- We do not have access to each user's password, so we randomly generate a password for each user we scrapped from IMDB. The password is a random combination 5 to 10 numbers and characters.

- Some actors' birth date cannot be achieved from their detailed page. We leave them as NULL.
- The gender of the movie stars is not explicitely listed on their detail page. However, by seeking their role in the movie (i.e, actor/actress), we can acquire their gender.

3.4 Sample Queries

3.4.1 User related

3.4.2 CRUD on movies, actors and directors

3.4.3 Comments related

3.5 Data Analysis

4 Future Works

5 Result

6 Conclusion

In this project we build a moive database based on a large amount of data collected from **IMDB**. Additional attention is paid to assure a **BCNF** form is kept for every entity in our database, thus the redundancy part is eliminated. We also **implement useful functions** for our users to search for moives, casts or other users using name, time information, rate(only moives) and genres(only moives) as the keyword.

In order to improve user experience and replace the complex database operations by a few simple mouse clicks from users, we build a user friendly website. Based on that, abundant **personalized interaction behaviors** can be achieved, which serves our users more conveniently. We hold an integral process of account management for users. Users can add comments to a moive and rate on it as well as discover their approbated reviews or users.

It is common for a moive database to recommend moives that the users have potential interest in. However, how to make a personalized recommendation lists of high quality is always a problem. We use a lot of techinques in the data analysis to satisfy the needs from users . We build a **Restricted Boltzmann Machines** with **Neighbourhood model** and get gratifying results.

7 Self Evaluation

In this part, we evaulate the advantages and disadvantages of our project/

7.1 What we have achieved

1. Build a database conforming to the low redundancy paradigm
2. Implement and optimize many necessary searching functionss
3. Create a user-friendly website to lower the bar of use
4. Feed the database with a suitable amount of data and design a proper data analysis algorithm to satisfy personalized requirements

7.2 Future Improvements

1. More information about moives, dierctors, actors can be added to the database. For example, for actors we may add their nationality and oscar nominations; for movies we may add its budget and the classification (PG13, NC17).

2. More interaction functions can be add, like ′like" a moive or ′subsrcibe" a user

3. In the future, we plan to distiguish the users of our platform. We will allow a common user to upgrade as administrator, which will reduce the effort to maintain and supervise the comments other users post.

4. Currently the web crawler is implemeted using synced requests, which reduced its performance. In the future, we will write the crawler using async requeststst to speed up the crawling performance and acquire more data for our database.

8 Contribution