Changlei Li, Sara Liu, Winston Tang, Jack Yu
Professor Susan Davidson
CIS550 Database and Information Systems
Final Project Report
April 27, 2022

The Movie World

1. Introduction

1.1 Problem statement
The social distancing life mode during pandemic period triggers a highly increasing demand of indoor entertainments. Watching movies is one of the best choices to help people relaxing and releasing stress. A movie exploration website is therefore needed to help people searching for what they want to watch and also suggest popular movies.

1.2 Deliverables
The website provides several useful searching tools to meet different user needs, including movie or a specific actor/actress search, exploring most popular movies and high-quality movies. If user does not know what they want to watch, the application also could recommend movies through random selection strategy. In addition, statistics data is provided for users who are interested in what's trending in movies now.

List of webpages:

- Summary page: show recent highlights, and statistics data of movies
- Webpage for movie search and actor/actress search
- Page of Editor's Pick: users only input limited information or even none, the search engine with designed algorithms would do the hard work to return popular and high-quality movies

1.3 List of group members

- Winston Tang - wyztang@seas.upenn.edu - winstonyz
- Changlei Li - changlei@seas.upenn.edu – changleipenn
- Jack Yu - jackyu@seas.upenn.edu - jack50042
- Sara Liu - liumengq@seas.upenn.edu – mengqisara

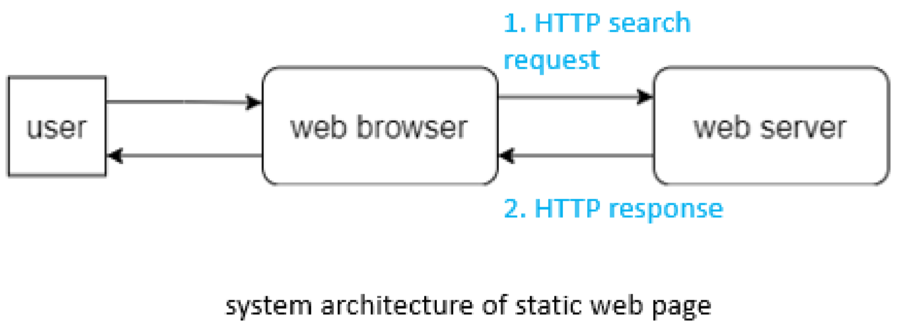
2. Architecture

MySQL database is used for this project. JavaScript, Node.js, React, Express are used for frontend implementation **(Figure 1)**.

This movie exploration system will be read heavy since users will search for movies and other information more frequently than write a review or a rating (this function is not supported though). To reduce the latency and maintain high availability for good user experience, static web technology is used to provide statistics data which does not need to be real time data and could be updated once a week.

Other functions like search for movies based on given criteria need to be supported dynamically. Thus, the API gets the data from database based on input value (see dynamic web page graph below).

For the Editor's pick functions, limited user input is given. Therefore, those selections could be done upfront, and the intermediate selection results could be saved to help reducing the query time and rendering HTTP response much faster. More details are given in performance evaluation session.



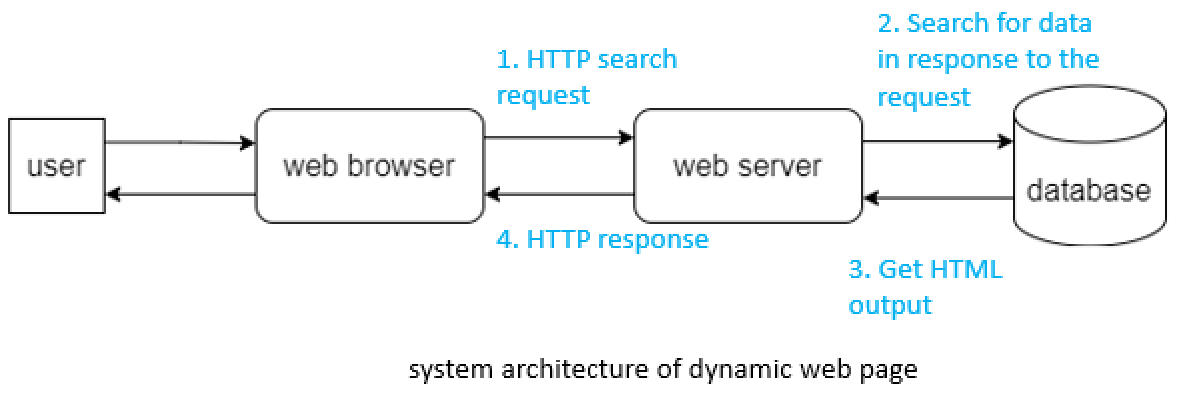


Figure 1. Flowchart of our application realization

3. Data and database

We wanted to create an application about movies, so 1) (requirements analysis) useful datasets about movies were obtained mainly through google dataset search and Kaggle. 2) (conceptual database design) Looking upon the existing datasets that were collected, entities resolution was done. This is done by thinking about what functions we wanted to implement with this application. An Entity Resolution (ER) diagram **(Figure 2)** was created to help facilitate the development of our database. 3) (logical database design). Based on the previously created ER diagram, a schema for each table was obtained. 4) (physical database design) performing data ingesting and building tables and setting

3.1 Datasets description and ingestion procedures

name.basics.csv - https://datasets.imdbws.com/name.basics.tsv.gz
Contains the following information for names:

- nconst (string) - alphanumeric unique identifier of the name/person
- primaryName (string)– name by which the person is most often credited
- birthYear – in YYYY format
- deathYear – in YYYY format if applicable, else '\N'
- primaryProfession (array of strings)– the top-3 professions of the person
- knownForTitles (array of tconsts) – titles the person is known for

Cleaning done:

- remove entries with birthYear (\N)
- parse and unwind the primaryProfession into string values
- parse and unwind the knownForTitles into string values
- filter actor and actress based on primaryProfession column
- separate out the actor and actress into individual tables
- the actors associated with nconst and tconst as a separate table called actorBy and actor
- the actress associated with nconst and tconst as a separate table called actress By and actress

title.basics.csv - https://datasets.imdbws.com/title.basics.tsv.gz
Contains the following information for titles:

tconst (string) - alphanumeric unique identifier of the title
titleType (string) – the type/format of the title (e.g. movie, short, tvseries, tvepisode, video, etc)
primaryTitle (string) – the more popular title / the title used by the filmmakers on promotional materials at the point of release
originalTitle (string) - original title, in the original language
isAdult (boolean) - 0: non-adult title; 1: adult title
startYear (YYYY) – represents the release year of a title. In the case of TV Series, it is the series start year
endYear (YYYY) – TV Series end year. ‘\N’ for all other title types
runtimeMinutes – primary runtime of the title, in minutes
genres (string array) – includes up to three genres associated with the title

Cleaning done:

- filtered region with “US”
- Removed entries with startYear(\N)
- filtered titleType with “movie”
- removed originalTitle, titleType, and endYear
- separate out the genres associated with tconst as sparate tables called genreOf and genre

title.crew.csv - https://datasets.imdbws.com/title.crew.tsv.gz
Contains the director and writer information for all the titles in IMDb. Fields include:

tconst (string) - alphanumeric unique identifier of the title
directors (array of nconsts) - director(s) of the given title
writers (array of nconsts) – writer(s) of the given title

Cleaning done:

- parse and unwind the directors' array and writers' array into string values
- separate out the writers associated with nconst as a separate table called writerBy and writer
- separate out the directors associated with nconst as a separate table called directedBy and director

Title.ratings.csv - https://datasets.imdbws.com/title.ratings.tsv.gz
Contains the IMDb rating and votes information for titles

- tconst (string) - alphanumeric unique identifier of the title
- averageRating – weighted average of all the individual user ratings
- numVotes - number of votes the title has received

Cleaning done:

- remove unnecessary columns numVotes

3.2 The links to the datasets are:

https://www.imdb.com/interfaces/

https://datasets.imdbws.com/

https://www.themoviedb.org/

https://grouplens.org/datasets/movielens/

3.3 ER diagram

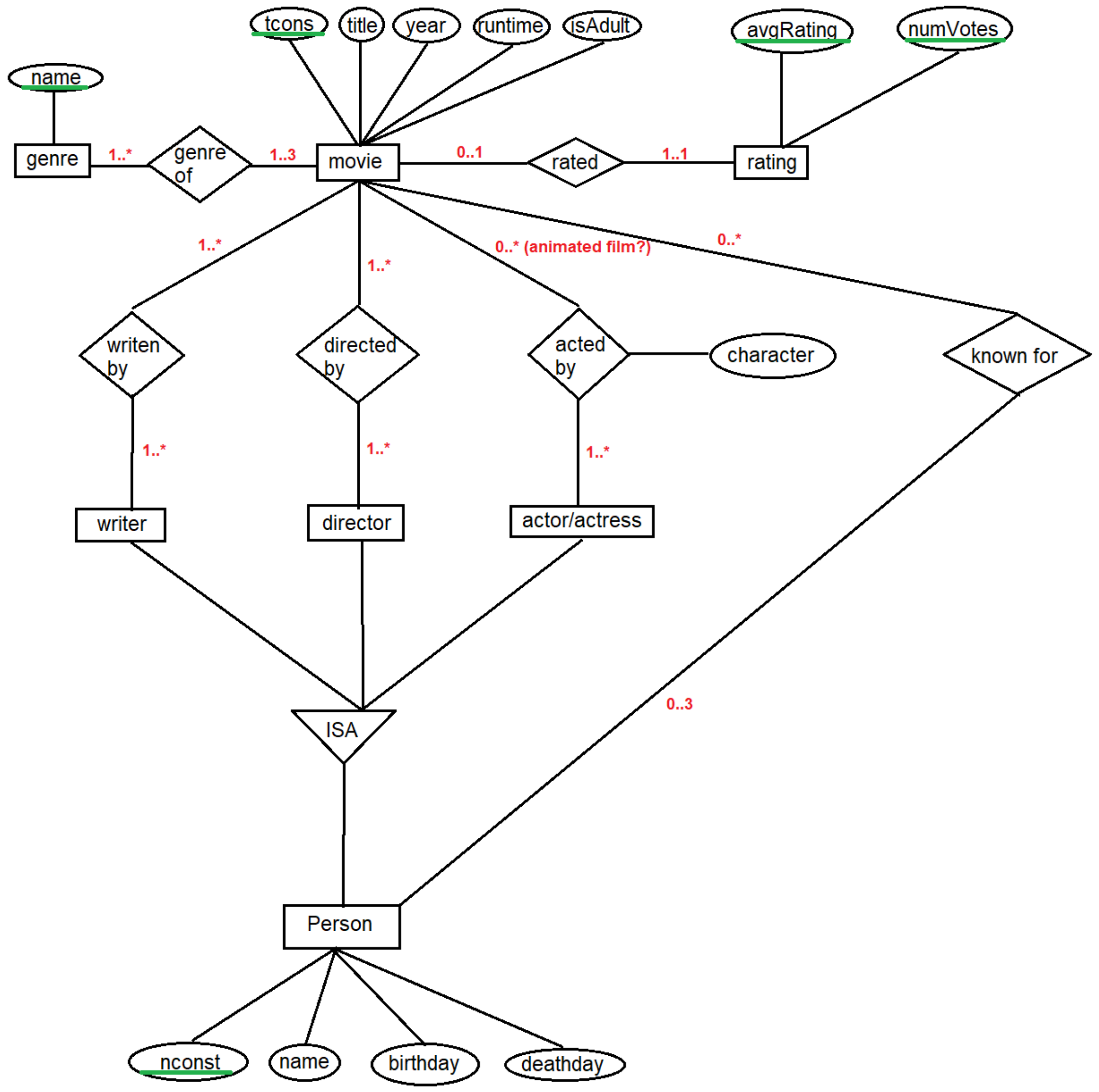


Figure 2. Entity resolutions (ER) diagram for the movie world database

3.3 Tables

Tables are created based upon required functionalities of the application, including

- genre(name)
- genreOf(name, tconst), name FOREIGN KEY referencing genre(name), tconst FOREIGN KEY referencing movie(tconst)
- movie(tconst, title, year, runtime, isAdult)
- rating(avgRating, numVotes, tconst), tconst FOREIGN KEY referencing movie(tconst)
- knownFor(tcons, nconst), tcons FOREIGN KEY referencing movie(tcons), nconst FOREIGN KEY referencing person(nconst)

- person(nconst, name, birthday, deathday)
- writer(nconst)
- director(nconst)
- actor/actress(nconst)
- actorBy(nconst, tconst, character), nconst FOREIGN KEY referencing person(nconst), tcons FOREIGN KEY referencing movie(tcons)
- directedBy(nconst, tconst), nconst FOREIGN KEY referencing person(nconst), tcons FOREIGN KEY referencing movie(tcons)
- writtenBy(nconst, tconst), nconst FOREIGN KEY referencing person(nconst), tcons FOREIGN KEY referencing movie(tcons)

Tables movie, rating, person, writer, director, actor, actress provide entity data. knownFor, actorBy, directedBy, and writtenBy tables represent relations between entities. Table genre lists all movie types, and each tuple in genreOf table incorporates movieId (attr: tconst) and the corresponding movie genre (attr:name). With all abovementioned tables, we could be able to implement different APIs to respond to different user requests, i.e., exploring movies, searching for actors/actress, trending in movie, etc.

Details on the normal form used (3NF or BCNF).

All the tables are in the BCNF normal form, because "Each attribute shall refer to the key, the whole key, and nothing but the key. So help me Codd."

4. Web app description

This application has three major pages, besides the homepage. Homepage **(Figure 3a)** serves as a summary page showing highlights and some selected featured movies.

Movie search page **(Figure 3b)** incorporates different movie search criteria, i.e., year, rating, keyword, director, etc.. It enables several search methods to meet different user preferences. A wide range of search criteria provides robust search capability intending to accurately locking down to the information users look for.

Actor search page **(Figure 3c)** is designed for movie fans who are interests in knowing more information of a specific actor/actress. The database includes 150,000+ actors' information – so you do not want to miss the chance to understand more your favorite movie stars!

Last but not the least, we never want to disappoint our users and we understand that sometimes users just want some choices being put in front of them without putting too much thought into it. To that end, we created the Editor's Choice page **(Figure 3d)**, which provides three fantastic functions for people who is looking for creative ways to discover movies or could not find what they like using provided search criteria. The first function allows user to select a movie genre and a rating range. With this combined search conditions, it allows a range search without losing the accuracy. Second function needs even less input, only movie genre needs to be selected, and a complicated querying algorithm will be applied to find movies written or directed and played by highly rated writers, directors and actors for a specific type of movies. For the purpose of fun, a random movie pick function is also provided. Every time the user hits the button, a list of movies will be selected randomly. By removing the search constraints, you may be surprised by what's being returned.

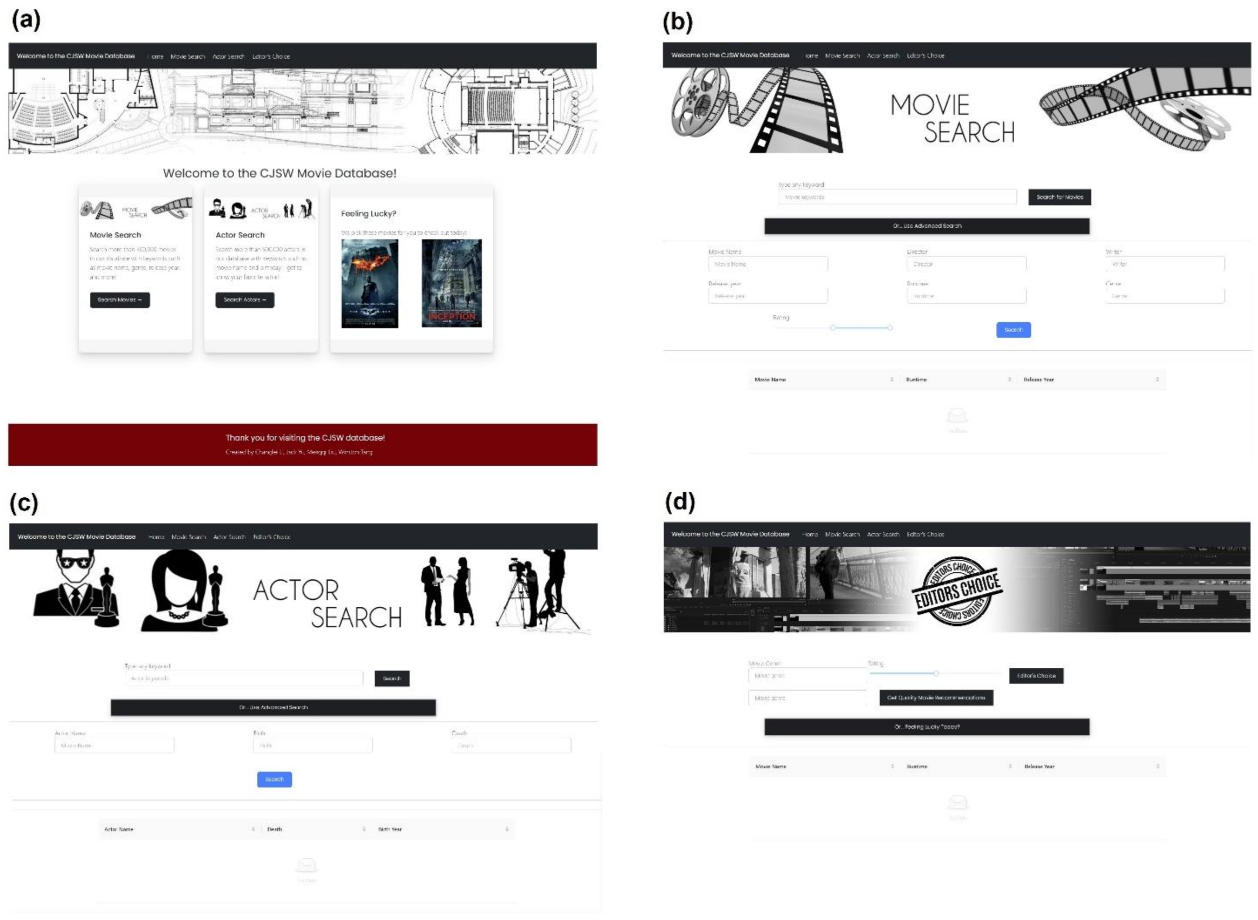

(a)
Welcome to the CJSW Movie Database
Welcome to the CJSW Movie Database!
Movie Search
Actor Search
Feeling Lucky?
Search Movies →
Search Actors →
Thank you for visiting the CJSW database!
(b)
MOVIE SEARCH
Search for Movies
Or... use Advanced Search
Movie Name
Runtime
Release Year
(c)
ACTOR SEARCH
Or... Use Advanced Search
Search
Actor Name
Death
Birth Year
(d)
EDITORS CHOICE
Editor's Choice
Get Quality Movie Recommendations
Or.. Feeling Lucky Today?

Figure 3. screenshot of the application a) homepage, b) movie search page, c)actor search page, d) editor's choice page

5. API specification

<u>getMovie API: a movie search API with different search criteria</u>

request path: /getmovie

request parameters: moviename/director/writer/actor/birthyear/deathyear/movierating /movieyear/runtime/genre response: tconst: movie id, title: movie title, year: movie release year, runtime: movie length, isAdult: flag for isAdult

sample request: http://127.0.0.1:8080/getmovie?field=moviename&fieldvalue=Kate

sample response: {"results":[{"tconst":"tt0035423","title":"Kate & Leopold","year":2001,"runtime":118,"isAdult":0},{"tconst":"tt1744793","title":"The Kate Logan Affair","year":2010,"runtime":85,"isAdult":0}]}

getActor API: rendering actor biographic data

request path: /getactor

request parameters: name/birthyear/deathyear

response: nconst: actor Id, name: actor name, birthyear: actor birth year, deathyear: actor death year

sample request: http://127.0.0.1:8080/getactor?field=name&fieldvalue=fred

sample response: {"results":[{"nconst":"nm0000001","name":"Fred Astaire","birthday":1899,"deathday":1987},{"nconst":"nm0000033","name":"Alfred Hitchcock","birthday":1899,"deathday":1980}]}

getPopularMovie API: a more powerful tool designed for search over a range for movies

request path: /getpopularmovie

request parameters: genre and rating range

response: tconst: movie id, title: movie title, year: movie release year, runtime: movie length

isAdult: flag for isAdult, genre: movie genre, avgRating: movie average rating

sample request: http://127.0.0.1:8080/getpopularmovie?rating=3&genre=Action

sample response: {"results":[{"tconst":"tt0118688","title":"Batman & Robin","year":1997,"runtime":125,"isAdult":0,"genre":"Action","avgRating":3.8},{"tconst":"tt0327554","title":"Catwoman","year":2004,"runtime":104,"isAdult":0,"genre":"Action","avgRating":3.4}]}

getRandomMovie API: Randomly pick movies without any search conditions. Being prepared for things unexpected but maybe exciting

request path: /getrandommovie

request parameters: none

response: tconst: movie id, title: movie title, year: movie release year, runtime: movie length, isAdult: flag for isAdult

sample request: http://127.0.0.1:8080/getrandommovie

sample response: {"results":[{"tconst":"tt3125182","title":"Kidney and Apple","year":2016,"runtime":94,"isAdult":0},{"tconst":"tt0362225","title":"Tell No One","year":2006,"runtime":131,"isAdult":0}]}

getQualityMovie API: using complex queries to filter highly rated directors, writers and actors for a certain movie genre and find movies in which they are devoted to

request path: /getqualitymovie

request parameters: genre

response: tconst: movie id, title: movie title, year: movie release year, runtime: movie length, isAdult: flag for isAdult

sample request: http://127.0.0.1:8080/getqualitymovie?genre=Action

sample response: {"results":[{"tconst":"tt0095840","title":"Payback","year":1990,"runtime":90,"isAdult":0},{"tconst":"tt0096817","title":"Angels","year":1990,"runtime":95,"isAdult":0}]}

6. Queries

Only major operations are listed here for selected queries. **See Appendix A** for more details.

Query to get movies to which highly rated actors/actress, director or writer are devoted. It is used to realize get high quality function in Editor's pick webpage

it uses movie genre, like "Action" as input, returns 100 high quality movies with title, year, runtime etc. Major operations are:

step 1: get movies in "Action" category and save the movie ids in the temporary table 'highRankingMovie'.

step 2: get an actor's movies in "Action" category and calculate average rating, save the actor's id in **temporary table 'highRankingActor'** if average rating is higher than a certain value. This operation is completed by **joining** three tables, using **average operation** and **selecting** entries meeting the rating threshold.

Step 3-5: repeat similar operations as step 2 to get **temporary tables 'highRankingActress', 'highRankingDirector', 'highRankingWriter'**.

Step 6-9: conduct JOIN operations to find movies by **joining between four temporary tables**, i.e., find movies acted by actors in 'highRankingActor' set and directed by directors in 'highRankingDirector' set by **joining** five tables.

Step 10: **union movie id sets obtained through step 6~9** and return a list of movies with return parameters as movie id, title, year, runtime and flag for isAdult.

<u>Query for results matching with a certain movie genre value and within a given movie rating range. This query is majorly for implementing get popular movie function in Editor's pick webpage</u>

Select movies with ratings in a certain range, i.e., 3.0 – 4.0 for a given genre, like "Action", join movie schema, rating schema and genreOf schema, order by number of votes in descending order, and project movie id, title, year, runtime, flag for isAdult, genre and rating for first 100 tuples.

<u>Query for movies based on selected actor/actress. This query is to realize movie search function in movie search webpage</u>

Step 1: select actors with their names matching with search parameter, join person schema, actorBy schema, and movie schema and project movie id, title, year, runtime, flag for isAdult and actor name.

Step 2: select actress with their names matching with search parameter, join person schema, actressBy schema, and movie schema and project movie id, title, year, runtime, flag for isAdult and actor name.

Step 3: union result sets obtained from step 1 and 2 and return first 100 tuples.

<u>Query for movies performed by or directed by or written by actor/actress, directors or writers whose birthyear matching with search parameter. This query enables searching movie by birthyear function in movie search webpage</u>

Select people with matching birthyear from person schema, join with directedBy, written by, actorBy, and actressBy schemas to obtain corresponding movie ids, then join with movie schema to project movie id, title, year, runtime, flag for isAdult and person name and the return results are limited to 100 tuples.

<u>Query for movies directed by a given director name. This query enables searching movie by director function in movie search webpage</u>

Select people with name like search parameter from person schema, join person schema, directorBy schema, and movie schema to project movie id, title, year, runtime, flag for isAdult and person name as directorName and return 100 tuples.

7. Performance Evaluation

A summary table **(Table 1)** of optimizations conducted to reduce complex query cost and latency of response to client request:

Query No.	description	Initial cost	Issues	Solutions	Time after optimization	Comments
1	Search for movies for a certain genre and rating range	4s348ms	wildcard string search, i.e.,'%Action%' is used; index not applicable	Partition and cache movie info for different genres	<1s (-77%)	No latency felt when responding to client request
2	Search for movies involved highly rated actors, directors, etc..	7s880ms	Inappropriate projection of attributes	Project needed attrs	4s570ms (-42%)	
			Wildcard string search, full table scan for selection	Cache search results	2s843ms(-22% time)	No latency felt when responding to client request
			Repeated nested loop join	Save intermediate JOIN results	No much change	Next step: refer to explain plans to understand why not working and explore other solutions

Table 1. optimization summary

Detail optimization steps and learnings are provided in Appendix B

8. Technical Challenges

The workload on hardware resources – since we are working with the AWS MySQL database in the free tier, the hardware resources are limited. Many of our classmates have reported they have been charged with an absurd amount of money, so we must be careful to incur fees during our databased development. We must keep the datasets uploaded to be minimum to avoid incurring a fee and to optimize the uploading time and quire performance. In addition, due to the foreign key constraints, the upload speed was extremely slow. This was overcome by adding the foreign key constraints to the tables after all the tables have been uploaded.

The development of database solution to match our application goal – during the entity resolution process, we have to consider multiple solutions to match our application goal, and how these tables would facilitate the formation of our functions or optimize the quire search without the need to join too many tables and at the same time taking the least amount of space to store those tables. This was overcome by considering multiple entity resolution plans and identifying the advantage and disadvantages of each plan, and eventually making a trade-off between them and selecting the best one that aligns with our application goal to implement for our database.

Massive data volumes – each of the datasets we have obtained easily contains millions of entries and they could not be opened with Excel, which would be easier to work with. In addition to the size is the time it takes to process the data. This was overcome by learning and using Jupyterlab and Pandas to do the data digestion.

We have also encountered some technical difficulties in front-end development. First of all, for styling and visual design, it took us quite a while to adjust small elements such as font sizes, button position and color choices – because we used packages such as 'antd', some of the default settings are hard to change or get by. Eventually with a flexible mindset and lots of online searches, we were able to deliver a front-end design that we see fit for the users. Additionally, we spent quite some time trying to connect the backend data to the front-end website. It requires a lot of details checking and thorough understanding of features such as props, State and Lifecycle. Small things such as spelling, different fields' corresponding properties names and table column data indexes and keys, sometimes can lead to minutes, if not hours of stagnation, development gridlock and debugging. Ultimately, with enough patience and online research, we were successful in bringing backend data live to the frontend website.

Appendix A:

Query to get movies to which highly rated actors/actress, director or writer are devoted. It is used to realize get high quality function in Editor's pick webpage

Operation details:

step 1: get movies in "Action" category and save the movie ids in the temporary table 'highRankingMovie'.

step 2: get an actor's movies in "Action" category and calculate average rating, save the actor's id in temporary table 'highRankingActor' if average rating is higher than a certain value. This operation is completed by joining three tables:rating, actorBy and genreOf and using group, average operations and selecting entries meeting the rating threshold.

Step 3: get an actress's movies in "Action" category and calculate average rating, save the actress's id in temporary table 'highRankingActress' if average rating is higher than a certain value. This operation is completed by joining three tables:rating, actressBy and genreOf and using group, average operations and selecting entries meeting the rating threshold.

Step 4: get a director's movies in "Action" category and calculate average rating, save the director id in temporary table 'highRankingDirector' if average rating is higher than a certain value. This operation is completed by joining three tables:rating, directedBy and genreOf and using group, average operations and selecting entries meeting the rating threshold.

Step 5: get a writer's movies in "Action" category and calculate average rating, save the writer id in temporary table 'highRankingWriter' if average rating is higher than a certain value. This operation is completed by joining three tables:rating, writtenBy and genreOf and using group, average operations and selecting entries meeting the rating threshold.

Step 6: find movies acted by actors in 'highRankingActor' set and directed by directors in 'highRankingDirector' set by joining five tables: highRankingMovie, highRankingDirector, directedBy, highRankingActor, ActorBy.

Step 7: find movies acted by actresses in 'highRankingActress' set and directed by directors in 'highRankingDirector' set by joining five tables: highRankingMovie, highRankingDirector, directedBy, highRankingActress, ActressBy.

Step 8: find movies acted by actors in 'highRankingActor' set and written by writers in 'highRankingWriter' set by joining five tables: highRankingMovie, highRankingWriter, writtenBy, highRankingActor, ActorBy.

Step 9: find movies acted by actresses in 'highRankingActress' set and written by writers in 'highRankingWriter' set by joining five tables: highRankingMovie, highRankingWriter, writtenBy, highRankingActress, ActressBy.

Step 10: union movie id sets obtained through step 6~9 and return a list of movies with return parameters as movie id, title, year, runtime and flag for isAdult.

Appendix B: Optimization Results and Discussions

Complex query 1: query for results matching with a certain movie genre value and within a given movie rating range

- Original query time: 4s348ms
- After-optimization time: <1s
- Root cause of high latency: wildcard string search, LIKE '%Action%' is used to search for target movie genre. This is challenging since it needs to detect presence of this string in any position within a column, with % at the beginning and the end, no index search could be used for data access and selection, therefore leading to low performance.
- Solution: caching technique is used to resolve the issue. genreOf schema is partitioned to 28 separate tables using different genre values, like Action, Comedy, etc. Therefore, selection operation for a certain movie genre is skipped since data entries are already stored in corresponding movie genre schema.

Complex query 2: query to get movies to which highly rated actors/actress, director or writer are devoted

- Original query time: 7s880ms

First round optimization

- After-optimization time: 4s570ms
- Root cause of high cost: some not necessary attributes are kept. Some useful attributes are not projected initially, and additional step is used to retrieve those attributes, i.e., only movie id is projected when getting temporary table 'highRankingMovie', and additional round of traversal of movie schema is performed to get other movie attributes which increases the cost.
- Solution: get all needed attributes of movies and save them in temporary table 'highRankingMovie', remove non-necessary attributes when projecting results, i.e., genre name in 'highRankingDirector' and other temporary tables.

Second round optimization:

- After-optimization time: 2s843ms
- Root cause: wildcard string search, LIKE '%Action%' is used to select movies within that genre category. This operation is performed in five different query steps intending to get temporary tables 'highRankingMovie', 'highRankingActor', 'highRankingActress', 'highRankingDirector', 'highRankingWriter'. Performing a time consuming operation repeatedly could eventually induce high time cost.
- Solution: genreOf table gets partitioned and movie tuples are classified based on genre and stored in different tables, which could be used directly to retrieve movie tuples for a certain genre. This skips the full table scan, which is inefficient for locating matching movie tuples.

Third round optimization:

- After-optimization time: no further time saving from last round optimization

- Hypothesis: four sets are generated to store data of high reputation actors, actresses, directors and writers. set 1: ‘highRankingActor’, set 2: ‘highRankingActress’, set 3: ‘highRankingDirector’, set 4: ‘highRankingWriter’. Then set 1 intersects with set 3, set 4 respectively, set 2 intersects with set 3, set 4 respectively as well. The resulting 4 sets are then merged. $(A\cap C)U(B\cap C)$ U $(A\cap D)U(B\cap D)$ is equivalent to $((AUB) \cap C)$ U $((AUB) \cap D)$. The hypothesis is to save AUB in a temporary table to save joining and merging cost due to repeated operations.
- Observation: no major change of time is detected.
- Next step: dig through into explain plans to understand cost of each step to detect other major time cost causes, and figure out why the hypothesis does not hold.