CPSC 437 Final Project

Tianyi Cai, Mengting Gu, Huaming Li, Di Wu, Xin Yan, Jun Zhao

Abstract

IMDb is an online database of information related to films, television programs, and video games, including cast, production crew, fictional characters, biographies, plot summaries, trivia and reviews. Our project takes advantage of this resource to build our own database in SQLite, allowing for general queries and data analysis about movies, such as the publish dates and genre relationships. We have also built a web interface for visualization of the results.

1 Introduction

Background

We all love watching movies, yet we have rarely take a look about the meta-data about the movies as a whole. IMDb provides great database where it has collected more than 3,500,000 titles including TV series, movies and documentaries, etc, plus information of about 3 million actors and actresses.

Motivation

Although we can easily do a single query on the IMDb database about a particular movie, we cannot visualize all the statistics about whole movies data. For our purpose, we decided to build up our own database, which only contains the movies collected in IMDb, excluding all the TV series and others to make our data clean and keep our focus on the most interesting collection.

Functionality design

Our goal is to create our own database from the IMDb resource which allows queries for individual movies and meta-query for general statistics. We aim to create a user-friendly web interface and allows users to input query though clicking rather than command-line input. We have also worked to demonstrate our analysis result through various data visualization methods to provide a clear and better experience.

2 Database construction

Overview

- Download list files from IMDb ftp site.
- Use imdbpy2sql.py script in the IMDbPY module to construct original vast database in sqlite3 from list files.
- Change model.py in Django, add table schemas.
- Migrate changes to databases.
- Select desired information from original giant database and insert into new database constructed by Django.

Introduction to Django

Django is a high-level python web framework that allows rapid development and clean, pragmatic design. It?s designed to help developers take applications from concept to launch with in-built frames. It is a open-source software easily downloadable from https://www.djangoproject.com/download/.

Raw data collection

We have found data online from IMDb database information about movies (including the published dates about movie, genre, actor and actress information, etc.) and actors/actresses (name, cast information, etc.). The plain text file can be found at http://www.imdb.com/interfaces. In our project, we use a tool imdbpy2sql from the python module IMDbPY (http://imdbpy.sourceforge.net/) to build an initial giant database in SQLite. Table in this database include:

`company_type, role_type, company_name, char_name, cast_info, comp_cast_type, name, movie_companies, info_type, link_type, keyword, person_info, movie_info_idx, aka_name, movie_keyword, kind_type, aka_title, complete_cast, title, movie_link, movie_info.`

Information needed will be copied from the original database to our database used in Django, which will be created by the models part of Django. We have built our database, also web interface based on the server provided by Koding. But due to the limited space allowed for an account, we put only tables that are used for later analysis and visualization on Koding. More database are built off-line on our local computer.

The following tables from the original database are processed and desired information is copied to our database used for visualization.

- title:
  id | title | imdb_index | kind_id | production_year | imdb_id | phonetic_code | episode_of_id | season_nr | episode_nr | series_years | md5sum
  We use id (refers to movie_id in other tables) and title for movies.

- movie_info:
  id | movie_id | info_type_id | info | note
  We use id, movie_id, info, with info_type_id = 3.
  *In the table info_type, each into_type_id is related to a kind of information. Here info_type_id = 3 refers to the information of "release dates".

- movie_info_idx:
  id | movie_id | info_type_id | info | note
  We need id, movie_id, info, with info_type_id = 101. Here info_type_id = 101 refers to information of "ratings".

- name:
  id | name—imdb_index | imdb_id | gender | name_pcode_cf | name_pcode_nf | surname_pcode | md5sum
  We need id (refers to person_id in other tables), and name, gender for actor/actress.

- cast_info: id | person_id | movie_id | person_role_id | note—nr_order | role_id
  We need id, person_id, movie_id to record the relationship between movies and actor/actress.

Database Structure

In this part, table schemas in our database will be described in detail, including SQL code to get information from the original database.

In order to copy records from a database to another in SQLite, first we need to attach the new database to the original one:

```
attach 'imdbs.db' as db2;
```

Tables in our database include (primary key marked with a *):

- movies_dates: (id*, movie_id, date)
  Include information on release dates of movies. Same movie may have multiple release dates, so each line has a unique id rather than movie_id. This table will be used in the "Movie Calendar" visualization.

  The original data in this table is formatted as: "Country: Two Digits Day Abbreviated month name Four Digits Year", for convenience of future analysis, a custom script "ChangeDate.py" is used to re-format this attribute. In the script, the built-in

shell of Django is used to access and update data items. There are also some tuples with incomplete information, like only year, or only year and month, which will be deleted from the database. There are 4130369 records before re-formatting, and after that, 473009 queries are deleted. SQL query to retrieve these data is:

```
insert into db2.movies\_dates(id,movie\_id,date)
select id, movie\_id, info from movie\_info where info\_type_id=16;
```

- **movies_ratings: (movie_id*, title, rating)**
  Include title and rating information of different movies. This table will be used in the "Genre Graph" visualisation, for picking top rating movies.

```
insert into db2.movies\_ratings(movie\_id,title,rating)
select title.id, title, info from title, movie\_info\_idx
where title.id=movie\_info\_idx.movie\_id
and movie\_info\_idx.info\_type\_id=101;
```

- **movies_genres: (id*, movie_id, genre)**
  Include genre information of movies. Same movie may have different genres, so each line has a unique id rather than movie_id. This table will be used in the ?Genre Graph? and also ?Word Cloud? visualisation, for getting genre information for each movie. Unfortunately, some movies are lacking information of genres. For these movies, we will randomly assign a genre from all possible genres in the analysis.

```
insert into db2.movies_genres(id,movie_id,genre)
select id, movie_id, info from movie_info
where info_type_id=3;
```

- **movies_person: (person_id*, name, gender)**
  Include actor and actress information, their id, name and gender. This table will be used in the ?Word Cloud? visualisation, for retrieving person_id from a user input name.

- **movies_cast: (id*, person_id, movie_id)**
  Include cast information, each tuple records a movie_id and a person_id, representing that this person plays in the movie. This table will also be used in the ?Word Cloud? visualisation, for getting cast information of a specific person, then join with movies_genres table to retrieve genre information of these movies.

```
insert into db2.movies_ratings(movie\_id,title,rating)
insert into db2.movies\_cast(id, movie\_id, person\_id)
select id, movie_id, person_id from cast_info;
```

Implementation

Front end

D3 graphics - HL

We use a JavaScript library called D3.js (data driven documents) for producing dynamic, interactive data visualization in our web browsers. It makes use of the widely implemented SVG, HTML5, and CSS standards. In particular, we implement 3 powerful visualization components using D3.js.

The first one is movie calendar. Given an input of specific data ranges from the user, we will output a calendar where the number of movies released each day is quantized into a diverging color. The values are visualized as colored cells per day. Days are arranged into columns by week, then grouped by month and years. This heat map representation will help the user to find some patterns when there would be large amount of movies released like Christmas, summer, etc.

The second one is genre graph. Given an input of integer n which means the first top n movies according to imdb.com rating, we will output a matrix diagram. We use an adjacency matrix to represent the network, where each cell ij represents an edge from vertex i to vertex j. Here, vertices represent movies, while edges represent how similar two movies are in genre. Given this two-dimensional representation of a graph, a natural visualization is to show the matrix. However, the effectiveness of a matrix diagram is heavily dependent on the order of rows and columns: if related nodes are placed closed to each other, it is easier to identify clusters and bridges. Here, we order the matrix so that similar movies are moved together and color depicts clusters computed by a community-detection algorithm.

The third one is word cloud. Given the name of a specific actor like Tom, Cruise, we will output a world cloud. Words are movies genres, and the importance of such genre is shown with font size, i.e., how many movies are in such genre for a given actor. This visualization will help the user to discover does genre mean a lot to a given actor - does he/she participate a movie because of the genre or something else?

Connection to backend

We use a view layer in Django to encapsulate the logic required to process a user's request and return the appropriate response to the user-end.

The view layer consists of two parts:

1. A Python module called "URLconf" (URL configuration) that maps URL patterns (simple regular expressions) to Python functions in the view module. a Python module called "views" that takes a Web request and returns a Web response, such as the HTML contents of a Web page, or a redirect, or a json object. The view contains the logic necessary to return the response.

   Specifically, when a user requests a page from a Django-powered site, the request is executed as follows:

(a) Django loads the Python module "URLconf" that specifies url patterns in regular expressions.

(b) Django goes through each URL pattern in sequence, and stops at the first hit that matches the requested URL.

(c) Django imports and calls the given view, which corresponds to a Python function in the "views" module. It passes the following arguments to the view function:

i. an instance of HttpRequest;

ii. if the matched regular expression returned no named groups, then the matches from the regular expression are given as positional arguments;

2. Otherwise, if no regular expression is found match, or if any exception is raised in the meantime, Django invokes an built-in error handling view.

Webpage

We use Hyper Text Markup Language (HTML) for describing web documents. Besides, Cascading Style Sheets (CSS) are used to style and format the HTML elements. The CSS is implemented in an external styling fashion, where the style attributes are stored in external CSS files. To achieve superior visual exhibition of webpages, we adopt HTML- and CSS-based design templates from Bootstrap ofr typography, forms, buttons, navigations and other interface components.

The overall design of our webpage architecture is as follows:

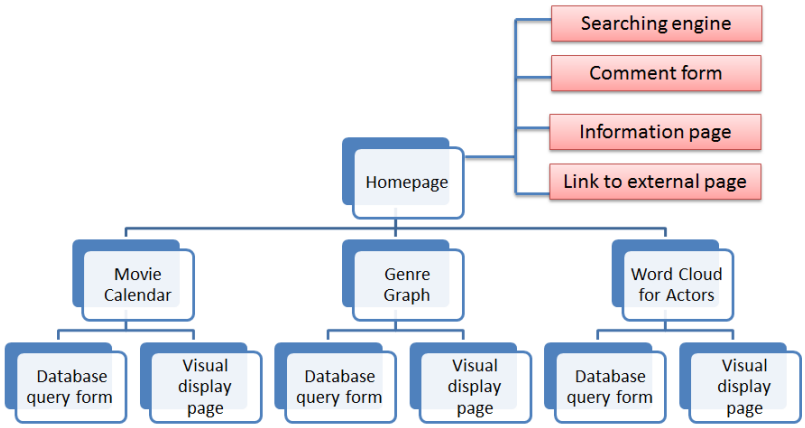

Figure 1: Design of webpage architecture

Back end

We take Modelviewcontroller software architectural pattern to implement our web application. It divides our web application into three interconnected parts, so as to separate internal representations of information from the ways that information is presented to or accepted from the user. We choose Django framework to develop our data-driven websites. It consists of an object-relational mapper which mediates between data models and a relational database ("Model"); a system for processing HTTP requests with a web templating system ("View"). Since Model and View have been demonstrated, here we talk about Controller.

Other than putting it in view, Controller part is implemented in api.py. In each of our functions, I created a corresponding api that looks at the relevant models and fetches data from database through model. Then api filters the data according to user input provided by front end. The model is also modified when join operation is needed. Last, api organizes the desired data and returns it to view.

3 Results

In the current project, we design three interactive database query and visualization functionalities: a movie calendar tool that allows user to input a date range for query, and returns a heat map that display the movies released in the range of query; a genre graph tool that takes a user-defined number of top ranking movies as input, and generate a matrix-like graph that clusters the movies according to their genres, and display the genre clusters in a block-diagonalized form; a word cloud graph for a movie actor specified by the user, where the most represented genre cast by that actor will be shown in bigger characters.

Movie calendar

Figure 2 shows a movie calendar displayed in a heat map form. A heat map is a graphical representation of data where the individual values contained in a matrix are represented as colors. Each day in the queried date range is represented as a small, discrete square, and is color-coded according the number of movie releases on that day. The movie calendar gives a very compelling and instantaneous representation of the movie releases, and can be used for movie producers to determine the optimal timing to release their movies.

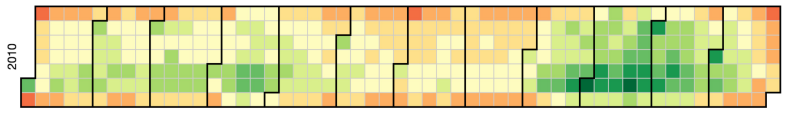

Figure 2: Example of movie calendar heat map

Genre graph

For n = 50, we get a matrix diagram below. We see that there are two main clusters purple and green.

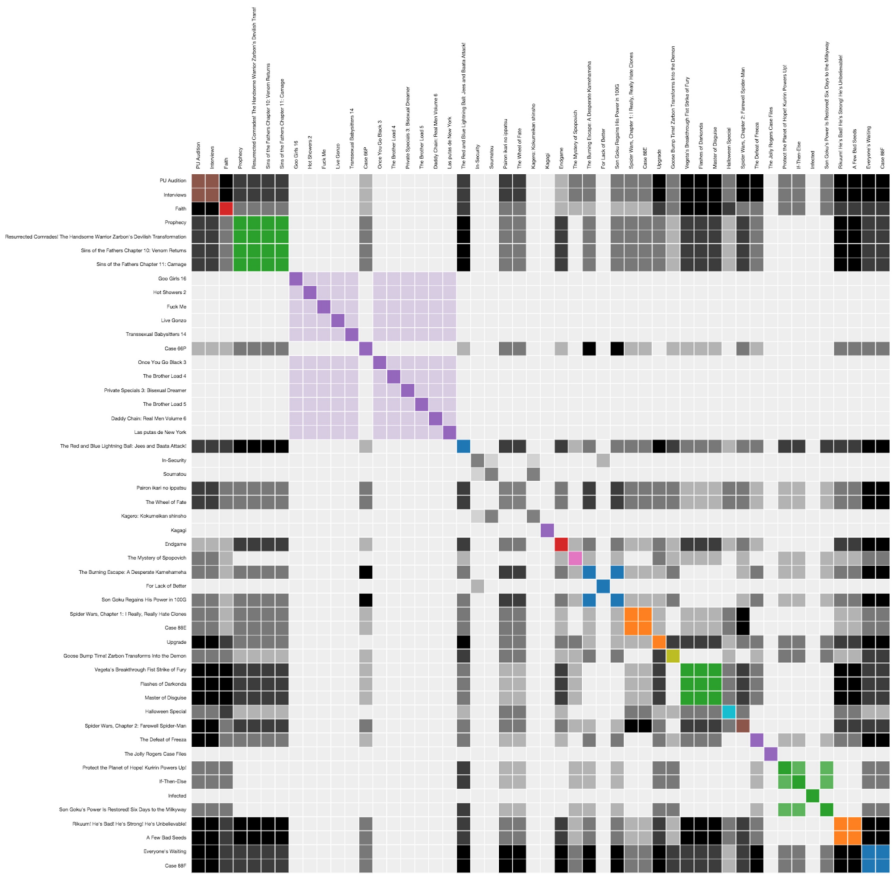

Figure 3: Matrix diagram with $n = 50$

Word cloud

For actors name Tom Cruise, we generate a word cloud below.

4 Future Directions

Up to now we have only used a few tables out of IMDB resources. There are many other tables that could be added from the lists. One direction can be incorporating the

Figure 4: Word cloud with actor's name Tom Cruise

actresses and actors data along with the cast information, which records each cast member in every film. Joining the cast table with the actors/actresses table will allow people to query for interesting questions like if several actors or actresses tend to perform together. This could also be linked to the genre table that we have constructed already to find out if performers tend to have preferences for movies of specific genres. Besides, IMDB has collected information about goofs in the movies which might be potentially interesting to attach to the search results if people are seeking for more fun in the movie.

Another direction goes to machine learning approach to dig for more information behind the data. For example, since we already have the information about the cast, genre and possibly ratings of the movies, connections among the movies and among the performers might be built to make inference, like making movie recommendations based on the search queries and results. Methods can also be used to learn the hot topics for movies and the general trend through ratings. Many more functions can be added based on peoples need.

5 Conclusion

We have built a movie database that support accurate queries for movies and bulk query for data analysis. In our project we used SQLite for backend database implementation and query processing, but other database softwares can also serve the purpose. Based on our database we have built a web interface for visualization of the queries.

Through this project, we understand better how the back-end database works for serving the user, especially through user-interactive interface. The transaction between a user's query to a query that the database software can parse and evaluate is not easy. Constructing a good database goes even beyond relation design database structure organization.