ECE 356 Project Report

Movies

Group 44

Group Members - Kartikeya Bhardwaj, Madhur Kumar Sharma, Harshit Manchanda

Introduction

We did a project on Movies. Our datasets are taken from one of the imdb datasets from Kaggle. Our client program has multiple functionalities taking inspiration from imdb and also including crud operation to add, delete and see movie details. Our code is structured into client, load-data and client side code. In the sections below we will go through the ER diagram, design decisions, relational schema, client and server side code and functionalities of our CLI.

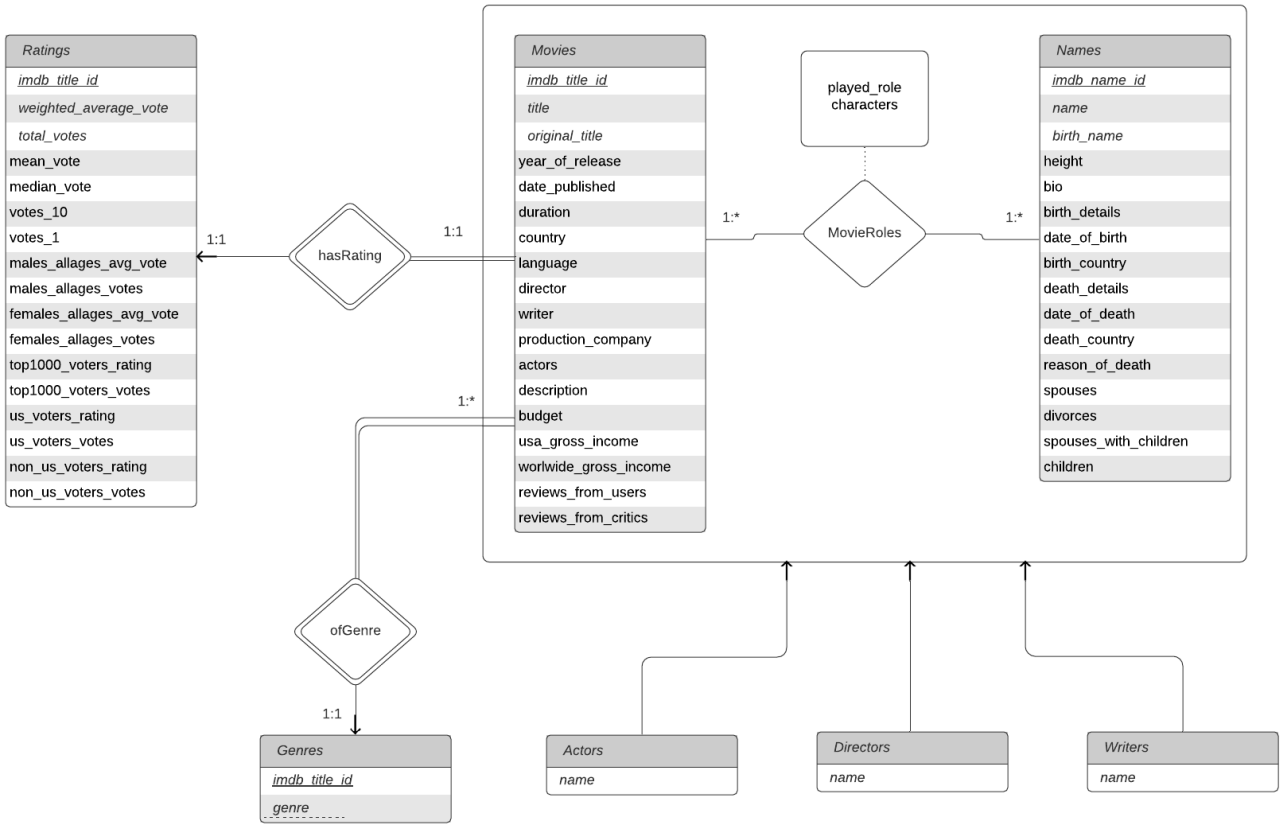
Github and Video Link

Code Repository: Since the project description asked for a Github link specifically, we have used personal accounts to host a repository on Github for this project. Please let us know if this is wrong, we can host a Gitlab repository too. Here is the link:
https://github.com/madhur4444/356Project/

Video Demo: One of the deliverables of this project is a video demo. We are hosting it here:
https://drive.google.com/drive/folders/14APdPcb03vSUuKKOA5jyZMM9ycsS5ddD?usp=sharing

ER Diagram

Here is our ER diagram -



The specialization of Actors, Directors and Writers are all partial specializations.

We will explain the design behind entity and relationship sets in the Design choices below.

Datasets and Design

Design choices and explanation

We took datasets from imdb movies dataset. We had to sanitize the data as there were some attributes which didn't make sense in our schema and we updated the inconsistencies of data using our code inside the src/sanitize-data-code/ folder.

From the Kaggle source, we were given four datasets as csv files, namely, Movies, Names, Ratings and movieRoles. However, this data was not workable for efficient querying in the scope of our project and we needed to sanitize it. For sanitization, we implemented the following design choices:

1. **Ensure movieRoles is a relationship between Movies and Names:** First, we used Python to check that the imdb_title_id (now referred to as “title id”) column and imdb_name_id (now referred to as “name id”) column are all referencing an existing title id in the Movies table and an existing name id in the Names table. If there were any rows that did not match this criteria, they were removed.

   This way, we were able to ensure that movieRoles has two foreign keys and thus, is a relationship set, as described in the ER diagram.

   Reason for this: We can then use the movieRoles table to quickly find out who the various people working for a movie are, like actors, directors, etc. instead of having to join two tables and manipulate data. This reduces the time of the operation significantly as this is an important use case (finding out the actors in a movie for example).

   This also helps us derive the various actors, directors, etc. in a movie with a view instead of three or four, or potentially n tables (depending on the future of this project). This makes it more scalable too.

2. **Remove unneeded columns from the Movies table:** There were a few columns in the Movies table that were representing data out-of-scope from this project, or represented duplicate data that was already elaborated in other tables. This included the “metascore” column that seemed more relevant for the company called IMDB rather than for the scope of this project. Moreover, avg_vote and votes were both columns that contained information already elaborated in the Ratings table. So, these three columns were removed.

   Reason for this: The user of our application does not need to know the metascore of a movie, as there are various critic reviews that can be obtained as well. Moreover, when we are collecting reviews, having duplicate data in the Movies table and also in the Ratings table increases the chance of errors as taught in this course. In order to keep the data as unique as possible, we removed the duplicate columns avg_vote and votes as well, which reduces the chance of errors.

3. **Creating weak entity sets:** The Ratings csv file consisted of the title id that was referring to the same column in the Movies csv file. Since there were a lot of attributes in the Ratings table, instead of almost doubling the number of attributes in the Movies table, we made the Ratings table to be a weak entity set of the Movies table. Since there is only one entry in the Ratings table for one movie, we do not need a discriminator column in the Ratings table itself.

Reason for this: Logically speaking, a rating is for a movie. Information about ratings for a movie cannot exist if the movie does not exist. That is why we made the Ratings table to be a weak entity set and depend on an entry in the Movies table. The reason we did not just move the attributes of the Ratings table as attributes of the Movies table is because firstly, this would dramatically increase the number of attributes of the Movies table compared to the rest of the tables, and secondly, having a weak entity set helps keep all information regarding ratings of a movie in a separate set, while still depending on the original set. This way, adding more information about the ratings of a movie can be done in a more modular fashion which helps with the future of this database (removing this information is much easier).

The Movies table also had an attribute called genre that included various genres of a movie, stored as one string. This created quite messy data and we believed it could be stored in a manner that was more scalable. So, we created another weak entity set for the genre attributes as the Genre table. Here, the genre attribute (now in the Genre table) was the discriminator.

Reason for this: Logically speaking, a genre is of a movie. It cannot exist without a movie. So, having a weak entity set rather than a complete relationship makes sense in that respect. Secondly, by storing the genre as a row in the Genre table rather than as a string in the Movies table, adding and deleting genres is much easier, and more importantly, can be done directly on the sql side. A server does not need to pull existing data, manipulate the string and modify it in the table. It can just use SQL to insert or delete (or modify) rows in the Genre table. This makes it more scalable as well as easier to manipulate.

4. **Partial specializations of an aggregation:** The newly made relationship set movieRoles was made to easily access various types of people who worked for a movie, like actors and directors. So, we made partial specializations for Actors, Directors and Writers. Accordingly, we removed the actors, directors and writers columns in the Movies table as that had duplicate data now.

   Reason for this: The reason we used movieRoles for various types of people in a movie, instead of the columns in the existing Movies table, is because there was a column for name id in the movieRoles relationship set, or aggregation, that allowed us to display data of the types of people in much more detail. So now, instead of storing just names of actors, directors and writers in the Movies table, we could pull data of any type of role and much more detailed too by joining it with the Names table.

   Since we will be frequently using this, instead of having to write the entire query to pull data from movieRoles, we created views or specializations for actors, directors and writers. Anyone in the future who needs info about other roles can just create another specialization from this aggregation.

The reason all these specializations are partial is because an actor can also be a director or a writer, and vice versa. Just being a specific role for a certain movie does not stop a person from being another role for a different movie or even the same movie.

Primary/Foreign keys

PRIMARY KEYS:

Movies -> “imdb_title_id”
Names -> “imdb_name_id”
Ratings -> “imdb_title_id”
MovieRoles -> “imdb_title_id, imdb_name_id”
Genres -> “imdb_title_id, genre”

FOREIGN KEYS:

Movies: N/A
Names: N/A
Ratings: imdb_title_id -> Movies.imdb_title_id
MovieRoles: imdb_title_id -> Movies.imdb_title_id, imdb_name_id -> Names.imdb_name_id
Genres -> imdb_title_id -> Movies.imdb_title_id

Translating ER diagram to relational schema in SQL

Our ER diagram aptly describes the relational schema that we implemented and there is nothing in the ER diagram that was translated differently. Movies and Names table were sanitized to avoid null data, erratic data, etc. and the Ratings table was sanitized to be a weak entity set of the Movies table. So, the primary key of the Ratings table is a foreign key of the Movies table. movieRoles was sanitized to be a relationship set, so the title id and name id columns in movieRoles are both foreign keys to the primary keys of Movies table and Names table respectively.

The Genre table is a weak entity set of the Movies table. So, the title id column in the Genre table is a foreign key of the Movies table.

Actors, Directors and Writers are all views derived from the MovieRoles relationship set/aggregation.

Testing for the translation

We had created an SQL file called createTables.sql to create all the tables in the database with the appropriate columns, data types, primary keys and foreign keys. Now, in order to validate that the script had run successfully and the tables were created without any faults, we used the describe command in marmoset after running the script to validate that there were no errors in its execution. By doing that, we were able to evaluate the result of the script and make sure there were no errors before we loaded all the data into the tables from the csv files.

Once all the tables were created successfully and we checked that they were, we ran the load script called loadData.sql which takes all the data in the csv files uploaded to the ECE server and loads them into the tables that were created. This step is the most important in the whole project because without data being loaded properly in the tables, we cannot proceed further as we won't be able to query anything correctly which defeats the whole purpose. In order to verify that the data was loaded successfully, we first looked at how many rows were actually entered into a table and if that number was the same as the number of records in the csv file. Secondly, we looked at the warnings for the load command of every table and made sure that the errors weren't something that didn't let the data enter the table correctly and were something that could be ignored or fixed. We did encounter a problem where the data that needed to be entered in the MovieRoles table was not working correctly. Only 36 rows were being loaded which was not at all correct but after experimenting a little, we found the problem which was the line ending of the file and after changing it to "/r/n" we got a better result and fixed the problem. Lastly, we just wanted to make sure that even after all these checks, the data is correct and so we ran the SELECT query on all the tables to view if we could spot any abnormality but everything was correct and thus, we started implementing the client.

Sanitization

We sanitize the date_of_birth, date_of_death column in Names to reflect the common date of birth for all people associated with movies so we can efficiently query them.
We have to sanitize the country out of birth as there are some inconsistencies in data.
We have 7 entity sets and 3 relationship sets in our ER diagram. We have 52 attributes in total across entities and relations.

We also had to sanitize genres and MovieRoles as the data was not standardized and we had to manipulate it to make the standard genres by splitting and generalizing the data for Movies table to create new Genres and MovieRoles table.

Client And Server

Our client and server code is implemented in Python. We have client.py with main functions which give users the option to enter commands and server.py which parses the request, queries the database, sanitizes the response and sends it back to the client.

Ideal Requirements

Our ideal client should support various queries and things that the user/client wants to use.

Client -
Ideal client should support whatever query the user wants to enter and give an ideal response parsing what's important text that user writes and wants to know but ideally speaking it isn't possible.

It should be able to detect when a user has inputted wrong data and provide reasonable flexibility for "wrong" data (through data correction) as well.

Our client should be able to ideally provide a response to any question related to movies and also insert/select/update/delete any field in any table in our schema.

Our client should also ideally be able to prevent or stop any and all malicious attacks.

Server -
Ideal server should be able to process anything related to movies and query that it receives from client and query database to get the right response.

Actual client proposed

Since an ideal client will never be completed (in any deadline), we need to make compromises and reasonable assumptions about the major use cases of this client. The client we proposed will at least be able to handle user input in a reasonably secure manner. This means that a user cannot enter a name for a data field, cannot leave a field empty when it is not supposed to be empty, etc.

Secondly, our client needs to be able to at least create/update/delete rows in the Movies table. Based off of IMDB and Rotten Tomatoes, it is much more common to get information about movies than to have a movie with some missing information for ratings and person names. This is another compromise that we proposed given our deadline.

Finally, our client needs to be able to gather the top N movies, or the bottom N movies, specific details about a movie, specific details about various actors, etc. as these are extremely popular use cases in existing client applications such as Rotten Tomatoes and IMDB.

Any other ideal requirement mentioned above cannot be implemented and is considered to be out-of-scope for this project, due to time constraints. This will be improved upon in the future.

Actual client implemented

Given the time constraints, it was not possible to implement an ideal client or even the best version client that we proposed so, we researched what are some of the most important things that a client should have to best utilize the Movies database. Based on what functionalities top websites like IMDB and Rotten Tomatoes provide to a user, we created a client that supports the basic CRUD operations which are create, read, update and delete along with popular and helpful query operations.

The picture below shows all the operations that the client cli offers to a user and there are a total of 17 commands. Here is a brief description of what each command does:

1. am - Add a movie to the Movies table. It takes an input for all the attributes present in the Movies table however, only the title is the required field. Rest can be empty and the user has the option to skip whichever field they want.
2. gmd - Gets all the movie details for a particular movie from the Movies table. It requests for a movie name and it needs to be exact since we are expecting the user would know which movie's details they want.
3. um - Update a particular attribute with a value specified by the user for a movie. The user enters a movie name and they are displayed all the attributes that the Movies table has and they can choose which column they want to edit and then input the new value to be stored.
4. dm - Delete a movie from the Movies table based on the movie name input from the user.
5. tm - Top x movies of all time based on their ratings stored in the Ratings table. The user inputs how many movies they want to see and based on that, the query returns only that many records.
6. actm - Displays all the actors and actresses that were in the movie entered by the user.
7. tmg - It displays the top x movies by genre. It takes 2 inputs, the number of movies that the user wants to view and the genre for which they want to see the top movies.
8. abt - It displays all the actors/actresses born on the current date (month and date). If the user inputs a date, it will display the actors born on that particular day of the month.
9. mvus - The x most voted movies in the US of all time which takes the number of movies the user wants to see as the input.
10. mvo - The x most voted movies everywhere of all time which also takes the number of movies as the input.

11. tmxc - This command displays the top rated x movies released in a particular country where the user supplies with the country for which they want to see the movies and also the number of movies that they want to see.
12. lrm - It returns the x most lowest rated movies of all time and the user inputs the number of movies that they want to view.
13. awaxy - It displays a list of all the actors and actresses that have acted in at least one movie between years x and y. The user enters the two years within which they want to see all the active people acting in the movies.
14. mba - This command returns all the movies in which a particular actor/actress has acted. The user inputs the name of the actor/actress for whom they want to view all the movies they have ever been in.
15. mdd - It displays all the movies directed by a director whose name is entered by the user.
16. tmy - This command displays all the top x movies of a particular year. The user is responsible for inputting the year and the number of movies they wanna view.
17. abmxy - It returns a list of the actors who are common in two movies. The user inputs names of two different movies and it returns all the actors who have acted in both of them.

Apart from all the commands mentioned above, we have two more options available to the user to help navigate the client better and offer help. Those are “h” which displays the list of all the commands if they want to view it again and the other one is “e” which is to exit from the client when they want to.

```
"am - Add movie",
"dm - Delete movie",
"um - Update movie data",
"gmd - Get Movie Details",
"tm - Top N Movies",
"actm - Actors in a movie",
"tmg - Top N movies By genre",
"abt - Actors born today (day and month)",
"mvus - N highest voted movies in the US",
"mvo - N highest voted movies in the world",
"tmxc - Top movies released in a country",
"lrm - N lowest rated movies",
"awaxy - Actors who were active in a timeframe (years)",
"mba - Movies acted by some actor",
"mdd - Movies directed by some director",
"tmy - Top N movies of a year",
"abmxy - Actors who acted in two specific movies",
"h - See commands list",
"e - Exit"
```

Test cases

In order to test the implementation of our client, we have two sets of tests. One is for the client responsible for checking the parsed user readable data based on the inputs from the user. It checks if a correct query is sent to the marmoset database and the response is exactly what is expected. The test cases for these have been implemented in the files named testClient.py. Another set of tests for the server have also been implemented in testServer.py to check the query sent to the server.py from client.py after the user inputs all the data so as to check the intermediate result returned by the SQL query generated by the server.

Server Tests

The test plan is to create sql queries in the valid and correct pattern by splitting the query returned by the client, send them to the server where the server will parse the request and return it. Once we get a result back, we compare it to the expected and actual result which is obtained by testing the query on the database itself on the ECE server.

Below is a picture indicating all the test cases for our server tests. The request is structured in such a way where we have a command in the beginning indicating the kind of operation requested by the user such as “am” which stands for adding a movie in the database. After that we have the arguments necessary to create the relevant query and they all are separated by a string “$$” which we obtain from the client. Server then parses this query, generates the SQL query with the correct syntax, sends it to the database and creates a dictionary with keys as the heading name and the value is a list with all the values of that column.

We have a total of 17 test cases as shown below:

```
testcases = [
    "am$$Spider-Man: No Way Home$$2021-12-16$$146$$USA, India, Canada$$English$$Jon Watts$$$$Marvel Studios$$Tom Holland, Ze
    "gmd$$Spider-Man: No Way Home",
    "um$$Spider-Man: No Way Home$$duration?=148",
    "dm$$Spider-Man: No Way Home",
    "tm$$5",
    "actm$$Miss Jerry",
    "tmg$$Drama$$5",
    "abt$$02-29",
    "mvus$$5",
    "mvo$$5",
    "tmxc$$USA$$5",
    "lrm$$5",
    "awaxy$$1910$$1911",
    "mba$$Asta Nielsen",
    "mdd$$Charles L. Gaskill",
    "tmy$$2010$$5",
    "abmxy$$Follie di jazz$$Cenerentola a Parigi"
]
```

The expected output of these tests are like this in sequence. Our tests check them to verify the server is acting fine and correctly processing, querying the database and returning parsed response fine. Here are the expected outputs for their respective commands in order:

```
expected = [
    True,
    {
        "Title": ["Spider-Man: No Way Home"],
        "Original Title": ["Spider-Man: No Way Home"],
        "Year": ["2021"],
        "Date Published": ["2021-12-16 00:00:00"],
        "Duration": ["146"],
        "Country": ["USA, India, Canada"],
        "Language": ["English"],
        "Director(s)": ["Jon Watts"],
        "Writer(s)": [""],
        "Production Company": ["Marvel Studios"],
        "Actor(s)": ["Tom Holland, Zendaya, Benedict Cumberbatch, Jacob Batalon, Jon Favreau"],
        "Description": ["Spider-Man: No Way Home is a 2021 American superhero film based on the Marvel Comics character Spider-Ma
        "Budget": ["$200,000,000"],
        "USA Gross Income": ["$328,000,000"],
        "Worldwide Gross Income": ["$751,300,000"],
        "Number of User Reviews": ["1297"],
        "Number of Critic Reviews": ["327"]
    },
    True,
    True,
```

```
{
    "Title": ["Suvarna Sundari", "Lejos de Casa pelicula Venezolana", "Jeeudo", "Ek", "Notuku Potu"],
    "Rating": ["9.90", "9.80", "9.80", "9.80", "9.80"],
    "Language": ["Telugu, Kannada", "Spanish", "English, Nepali", "Telugu", "Telugu"],
    "Duration": ["120", "87", "150", "129", "121"]
},
{
    "Actor Name": ["Blanche Bayliss", "William Courtenay", "Chauncey Depew"]
},
{
    'Name': ['Lejos de Casa pelicula Venezolana', 'Ek', 'Jeeudo', 'Hopeful Notes', 'Isha'],
    'Rating': ['9.80', '9.80', '9.80', '9.70', '9.50'],
    'Language': ['Spanish', 'Telugu', 'English, Nepali', 'English', 'Malayalam'],
    'Duration': ['87', '129', '150', '94', '118']
},
{
    'Name': ['Joss Ackland', 'Dennis Farina', 'Antonio Sabato Jr.', 'Knut Agnred', 'Domingo Ambriz', 'Albert Augier', 'Cheste
},
{
    "Title": ["Il cavaliere oscuro", "Le ali della libertà", "Pulp Fiction", "Fight Club", "Forrest Gump"],
    "Votes": ["348363", "327264", "274101", "263329", "252635"],
    "Language": ["English, Mandarin", "English", "English, Spanish, French", "English", "English"],
    "Duration": ["152", "142", "154", "139", "142"]
},
{
    "Title": ["Le ali della libertà", "Il cavaliere oscuro", "Inception", "Fight Club", "Pulp Fiction"],
    "Votes": ["2278845", "2241615", "2002816", "1807440", "1780147"],
    "Language": ["English", "English, Mandarin", "English, Japanese, French", "English", "English, Spanish, French"],
    "Duration": ["142", "152", "148", "139", "154"]
},
```



```
{
    "Title": ["Hopeful Notes", "The Moving on Phase", "Love in Kilnerry", "Le ali della libertà", "As I Am"],
    "Rating": ["9.70", "9.50", "9.30", "9.30", "9.30"],
    "Language": ["English", "English", "English", "English", "English"],
    "Duration": ["94", "85", "100", "142", "62"]
},
{
    "Title": ["Prema Panjaram", "Holnap történt – A nagy bulvárfilm", "Seikai gûdo mooningu!!", "Ryûsei", "One Night: Choice
    "Rating": ["1.00", "1.00", "1.00", "1.00", "1.00"],
    "Language": ["Telugu", "Hungarian", "Japanese", "Japanese", "Min Nan, Mandarin"],
    "Duration": ["138", "82", "81", "78", "79"]
},
{
    "Name": ["Asta Nielsen", "Emil Albes", "Hugo Flink", "Gunnar Helsengreen", "Valdemar Psilander", "Mary Hagen", "Giuseppe
},
{
    "Title": ["Den sorte drøm", "Amleto", "La via senza gioia", "Fante-Anne"],
    "Language": ["", "German", "German", ""],
    "Duration": ["53", "131", "125", "93"]
},
{
    "Title": ["Cleopatra"],
    "Language": ["English"],
    "Duration": ["100"]
},
```



```
{
    "Title": ["Hopeful Notes", "Les Misérables in Concert: The 25th Anniversary", "Inception", "Doraleous and Associates", "A
    "Rating": ["9.70", "8.80", "8.80", "8.60", "8.60"],
    "Language": ["English", "English", "English, Japanese, French", "English", "Tamil"],
    "Duration": ["94", "178", "148", "100", "126"]
},
{
    "Name": ["Fred Astaire"]
}
```

Client Tests

Now in order to test the client side, we have similar test cases just like in the case of the server. The test plan for this is to create the query based on the format of command and arguments separated by the string $$ in the client using all the inputs entered by the user, send it to the server where it will create sql queries that are valid and correct by splitting the query returned by the client, send them to the server where the server will parse the request and return the result. On getting the result back, we compare it to the expected and actual result which is obtained by testing the query on the database itself on the ECE server.

The picture below indicates all the test cases for our client tests. The test case is structured in the same way as how a user would enter the data. Since the user first selects the command that they want to run, we have the command abbreviation in the beginning of every test case to indicate the type of operation. It is then followed by all the variables that are required to create the query which are entered by the user and since they press enter after every input, we have a “\n” between every argument. The client then uses this information to generate the query to pass to the server which then parses this query, generates the SQL query with the correct syntax, sends it to the database and creates a dictionary with keys as the heading name and the value is a list with all the values of that column and returns that to the client.

We have a total of 17 test cases one for each command and are as shown below:

```
testcases = [
    "am\nSpider-Man: No Way Home\n2021-12-16\n146\nUSA, India, Canada\nEnglish\nJon Watts\n\nMarvel Studios\nTom Holland, Zen
    "gmd\nSpider-Man: No Way Home\ne\n",
    "um\nSpider-Man: No Way Home\n3\n148\ne\n",
    "dm\nSpider-Man: No Way Home\ne\n",
    "tm\n5\ne\n",
    "actm\nMiss Jerry\ne\n",
    "tmg\nDrama\n5\ne\n",
    "abt\n02-29\ne\n",
    "mvus\n5\ne\n",
    "mvo\n5\ne\n",
    "tmxc\nUSA\n5\ne\n",
    "lrm\n5\ne\n",
    "awaxy\n1910\n1911\ne\n",
    "mba\nAsta Nielsen\ne\n",
    "mdd\nCharles L. Gaskill\ne\n",
    "tmy\n2010\n5\ne\n",
    "abmxy\nFollie di jazz\nCenerentola a Parigi\ne\n"
]
```

The expected output of these tests are like this in sequence. Our tests check them to verify that based on the input from the user, we generate a correct query to pass to the server and the server is acting fine and correctly processing, querying the database and returning parsed response fine. Here are the expected outputs for their respective commands in order:

```
expected = [
    [True],
    [{
        "Title": ["Spider-Man: No Way Home"],
        "Original Title": ["Spider-Man: No Way Home"],
        "Year": ["2021"],
        "Date Published": ["2021-12-16 00:00:00"],
        "Duration": ["146"],
        "Country": ["USA, India, Canada"],
        "Language": ["English"],
        "Director(s)": ["Jon Watts"],
        "Writer(s)": [""],
        "Production Company": ["Marvel Studios"],
        "Actor(s)": ["Tom Holland, Zendaya, Benedict Cumberbatch, Jacob Batalon, Jon Favreau"],
        "Description": ["Spider-Man: No Way Home is a 2021 American superhero film based on the Marvel Comics character Spider-M
        "Budget": ["$200,000,000"],
        "USA Gross Income": ["$328,000,000"],
        "Worldwide Gross Income": ["$751,300,000"],
        "Number of User Reviews": ["1297"],
        "Number of Critic Reviews": ["327"]
    }],
    [True],
    [True],
```

```
[{
    "Title": ["Suvarna Sundari", "Lejos de Casa pelicula Venezolana", "Jeeudo", "Ek", "Notuku Potu"],
    "Rating": ["9.90", "9.80", "9.80", "9.80", "9.80"],
    "Language": ["Telugu, Kannada", "Spanish", "English, Nepali", "Telugu", "Telugu"],
    "Duration": ["120", "87", "150", "129", "121"]
}],
[{
    "Actor Name": ["Blanche Bayliss", "William Courtenay", "Chauncey Depew"]
}],
[{
    'Name': ['Lejos de Casa pelicula Venezolana', 'Ek', 'Jeeudo', 'Hopeful Notes', 'Isha'],
    'Rating': ['9.80', '9.80', '9.80', '9.70', '9.50'],
    'Language': ['Spanish', 'Telugu', 'English, Nepali', 'English', 'Malayalam'],
    'Duration': ['87', '129', '150', '94', '118']
}],
[{
    'Name': ['Joss Ackland', 'Dennis Farina', 'Antonio Sabato Jr.', 'Knut Agnred', 'Domingo Ambriz', 'Albert Augier', 'Ches
}],
[{
    "Title": ["Il cavaliere oscuro", "Le ali della libertà", "Pulp Fiction", "Fight Club", "Forrest Gump"],
    "Votes": ["348363", "327264", "274101", "263329", "252635"],
    "Language": ["English, Mandarin", "English", "English, Spanish, French", "English", "English"],
    "Duration": ["152", "142", "154", "139", "142"]
}],
[{
    "Title": ["Le ali della libertà", "Il cavaliere oscuro", "Inception", "Fight Club", "Pulp Fiction"],
    "Votes": ["2278845", "2241615", "2002816", "1807440", "1780147"],
    "Language": ["English", "English, Mandarin", "English, Japanese, French", "English", "English, Spanish, French"],
    "Duration": ["142", "152", "148", "139", "154"]
}],
```

```
[{
    "Title": ["Hopeful Notes", "The Moving on Phase", "Love in Kilnerry", "Le ali della libertà", "As I Am"],
    "Rating": ["9.70", "9.50", "9.30", "9.30", "9.30"],
    "Language": ["English", "English", "English", "English", "English"],
    "Duration": ["94", "85", "100", "142", "62"]
}],
[{
    "Title": ["Prema Panjaram", "Holnap történt – A nagy bulvárfilm", "Seikai gûdo mooningu!!", "Ryûsei", "One Night: Choice o
    "Rating": ["1.00", "1.00", "1.00", "1.00", "1.00"],
    "Language": ["Telugu", "Hungarian", "Japanese", "Japanese", "Min Nan, Mandarin"],
    "Duration": ["138", "82", "81", "78", "79"]
}],
[{
    "Name": ["Asta Nielsen", "Emil Albes", "Hugo Flink", "Gunnar Helsengreen", "Valdemar Psilander", "Mary Hagen", "Giuseppe d
}],
[{
    "Title": ["Den sorte drøm", "Amleto", "La via senza gioia", "Fante-Anne"],
    "Language": ["", "German", "German", ""],
    "Duration": ["53", "131", "125", "93"]
}],
[{
    "Title": ["Cleopatra"],
    "Language": ["English"],
    "Duration": ["100"]
}],
```

```
[{
    "Title": ["Hopeful Notes", "Les Misérables in Concert: The 25th Anniversary", "Inception", "Doraleous and Associates", "Aa
    "Rating": ["9.70", "8.80", "8.80", "8.60", "8.60"],
    "Language": ["English", "English", "English, Japanese, French", "English", "Tamil"],
    "Duration": ["94", "178", "148", "100", "126"]
}],
[{
    "Name": ["Fred Astaire"]
}]
```


End

Thanks for reading our report, we spent countless nights trying to figure all this stuff out. Thanks for grading us and giving us the chance to do this project.