IMDB Data Management and Movie Recommendation System

Richard Joerger
Rochester Institute of Technology
raj2348@g.rit.edu

Yifei Sun
Rochester Institute of Technology
ys8800@g.rit.edu

Ajeeta Khatri
Rochester Institute of Technology
ak6038@g.rit.edu

Zhuo Liu
Rochester Institute of Technology
zl9901@g.rit.edu

1. OVERVIEW

This paper serves as an update of our current status on the project as well as a road map to our continued progress as further deadlines approach. In brief, up to this point we have designed our database, we have written the scripts to create the database and tables, we have begun entering data into said database, and have discussed our application design as we move forward. As part of this planning and status update, we have included our thoughts on how to correct our project goals if we have overestimated our abilities due to time and knowledge constraints but hopefully these fallback options need not be used.

2. DESIGN CHOICES

2.1 Requirements

Our first step while designing our application was figuring out what our requirements are. We sat down as a group and figured out what we wanted to accomplish. We figured out that a user should be able to either do a simple search, where they would search based on a single data feature, or choose from a set of more complex search types. An example of the simple search would be, searching for movies which contained an actor with a given name. This would return a list of movies where said actor has a credited role in the film, IE: all movies Nicholas Cage is in. An example of a possible complex query would be searching for all movies released between a range of years which were directed by a particular person and also featured a particular actor. For a complex query we would allow for any search value to be empty, if the value was empty we would interpret that to mean that the user wants any value for that field, thus making it a wild card. So as an example, if a user wanted to search for any movie starting Leonardo DiCaprio and was directed by Quentin Tarantino but they did not specify a year range, we would assume they want any year and we would effectively create a wild card search for that value that would return any movie regardless of the year.

2.2 Application Design

The next step in this process was figuring out how we would design the application itself. Our initial idea was to develop our application solely in Java using the MVC design pattern. This would not only be a tried and trusted approach, correctly separating the different layers of our application, it would also allow for us to better delineate tasks amongst ourselves. But, we realized quite quickly that none of us wanted to use Swing or JavaFX to create a GUI, especially because we all have very limited experience creating UIs using those libraries. The next logical step was a TUI but we believed that such a system would not be intuitive enough for a user. On top of this, trying to make it possible to preform complex queries seemed like far more work for ourselves. So ultimately we settled on using JavaScript to develop our front end which would make calls to an HTTP API which would handle our requests for getting data to and from the database. The server which would handle these API requests would be written in Java and would also serve as our database connector. The benefit of this approach is that we maintain our ability to use MVC. This enables us to better split the work load across all member of the team. Additionally, as a team we feel that this design plays to our team strengths as some of are team have experience working with JavaScript and the other part of the team has experience working with Java. Approaching the problem in this manner also gives us greater flexibility when it comes to how the user will interact with our database.

3. DATABASE AND RELATIONAL MODEL

3.1 ER-Model and database structure

All the tables and the constraints are described as follows:
users (uid,passwd,age,language) PK(uid).
history (uid,tconst,date) PK(uid,tconst) FK_1 (uid) FK_2 (tconst).
rating (uid,tconst,rating,isVote)PK(uid,tconst)FK_1 (uid) FK_2 (tconst).
general_movie (tconst,title,isAdult,type)PK(tconst).
genres (tconst,genre)PK(tconst,genre) FK(tconst).
localize (tconst,local-name,language,isOriginal) PK(tconst,local-name,language) FK(tconst).
tvSeries(series-tconst,isOver) PK(series-tconst) FK(series-tconst).
tvEpisode(episode-tconst,episode-number) PK(episode-tconst) FK(episode-tconst).
has(series-tconst,episode-tconst,season-number,broadcast-year) PK(series-tconst,episode-tconst). FK(series-tconst, episode-tconst)
movie(movie-tconst,release-year,runtime) PK(movie-tconst) FK(movie-tconst).

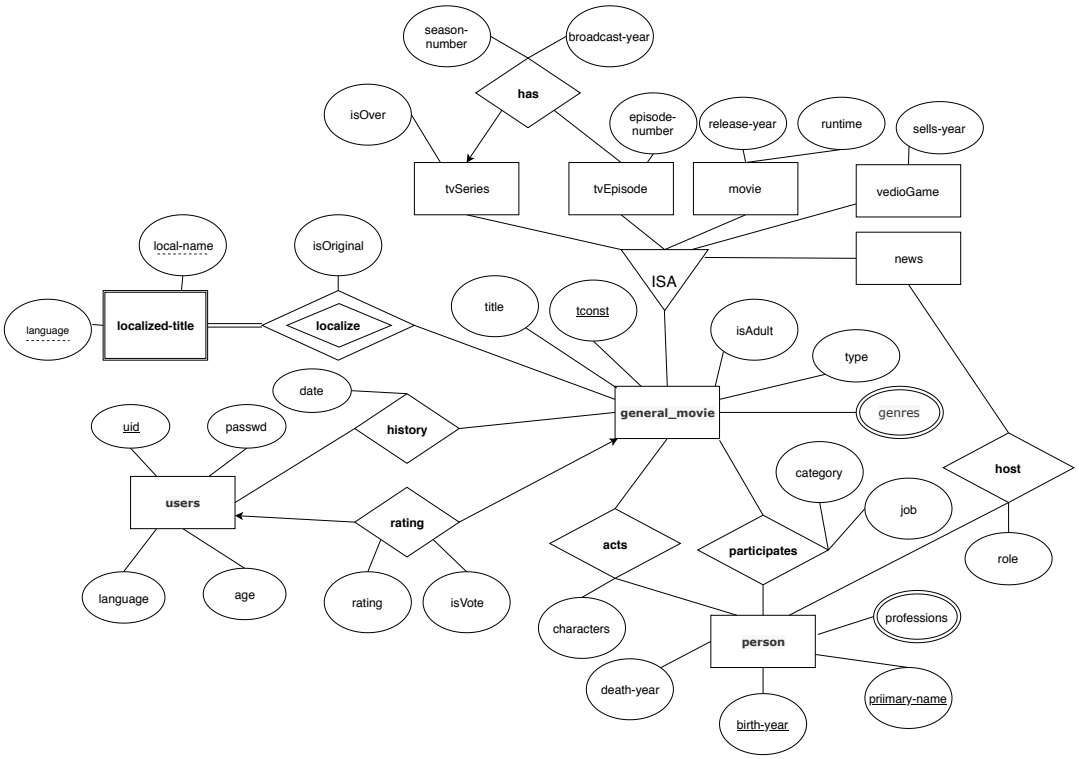

Figure 1: ER Model for our Application

videoGame(game-tconst,sells-year) PK(game-tconst) FK(game-tconst).
news(news-tconst) PK(news-tconst) FK(news-tconst).
persons(primary-name,birth-year,death-year) PK (primary-name, birth-year).
professions(primary-name,birth-year,profession) PK (primary-name, birth-year, profession) FK(primary-name, birth-year).
acts(act-name, birth-year, movie-tconst, character) PK (act-name, birth-year, movie-tconst, character) FK_1 (act-name, birth-year) FK_2 (movie-tconst).
participates(act-name, birth-year, movie-tconst, category, job) PK (act-name, birth-year, movie-tconst) FK_1 (act-name, birth-year) FK_2(movie-tconst).
host(news-tconst, host-name, birth-year, role) PK(news-tconst, host-name, birth-year) FK_1 (news-tconst) FK_2 (host-name, birth-year)

3.2 Raw data importing

In order to manipulate the IMDB data, we need to import the raw data from a .tsv file into our database. For this part, I choose to create tables in the same way that IMDB did. We create and maintain 'title_akas', 'title_basics', 'title_principals', 'title_crews', 'title_episodes', 'title_ratings' and 'name_basics' tables. We'll keep all the data along with every attribute even though there some of those attributes may have null values. Because we've kept all the data, we can manipulate these table and preform each query the way we want via SQL.

3.3 Raw data cleaning

Our data cleaning has two parts. The first step is to remove noisy data which has null values in a significant attribute column like 'birth-year' in 'persons'. The second step is to delete duplicated data.

What we got from IMDB is all raw data therefore we can not rely on its integrity and thus we must clean it. For instance, in the 'persons' table, our primary key is primary name and birth year. However, in the raw data, we may find some entries which do not have a birth year value. We will definitely remove this kind of data. At the same time, we assume that primary name and birth year can uniquely identify one person. So, when we have several tuples with same primary-title and birth-year, our solution is to record one of them in our database.

The way I do it is pretty straightforward. After we import all the raw data, we select the real data and generate a bright table with valid data. The data cleaning SQL script contains the query which will do the cleaning and where we store the data that is valid. Below is an example of a basic query which would remove all null values from the data.

SELECT primary-name, birth-year, death-year
FROM raw_names WHERE birth-year IS NOT NULL;

Now since we have some composite keys as primary keys in our databases, we have to clean all the duplicated data which has the same primary key attributes. For example, in the 'persons' table, we want birthYear and primary-name to be the primary key. We have to remove the data for which these two attributes have the same value. The following query shows how I removed the duplicate data.

CREATE TABLE new_data
SELECT * FROM 'name_basics' N1
GROUP BY (N1.primary-name, N1.birthYear);

Now, all we need to do is drop the old raw data and use

the cleaned data to generate our database and implement our application's functions.

Table 1: Timeline

Week 3	Phase 0	Submitted as a team
Week 4	Discussed the review from Phase0 and came up with an improved approach, discussed various scenarios and distributed the work	Team
Week 5	ER diagram, Functional Dependency, Initial design	Yifei and Zhuo : ER diagram, Richard - Initial Design, Ajeeta - Functional Dependency
Week 6	Database Implementation and Application backend - code to retrieve data from the database	Richard and Ajeeta: Java Application Zhuo and Yifei: Database Implementation, Insertion and creating views
Week 7	UI development and Integration	Zhou and Ajeeta: UI development Richard and Yifei: Integration of database with the Java application
Week 8	End-to-End Test(Verify that all goals are met)	As a team
Week 9	Submit Phase2	As a team

4. FUNCTIONAL DEPENDENCY ANALYSIS

For sake of brevity, we've only included our normalization approach for one of our tables since they all fundamentally work the same way. So in this example, we will work with the users relationship.

$users(uid, passwd, age, language)PK(uid)$

Our first step is to identify the the functional dependencies, for the user's table we identified a single dependency. We know that this is our only dependency since no other aspects of our relationship can be used to determine other attributes in the table. We know that this is the case for our relationship because all of our other attributes do not determine any of the other information represented in the relationship. Meaning, password, age, language or any combinations of those attributes do not imply any of the other information in the table. The only thing which has an implication on the rest of the data is the uid which leads us to our sole functional dependency. $FD : uid \rightarrow passwd, age, lang$ We know that this is a relationship because this is a 2D table, each column has a unique name, all cells hold a single value, all attributes are of the same domain, each row will be uniquely identifiable, and the order of rows and columns is irrelevant. Because we have a relation, we know our table is in first normal form. We can also deduce that this table is in second normal form because we only have one attribute as the primary key. This implies that all attributes are dependent on all of the keys which is the only additional requirement to assert that a relationship in first normal form is also in second normal form. From this we can also deduce that our relationship is in third normal form since it is required that a table's functional dependencies have no transitive dependencies. Since we only have one determinant we know that we cannot have any transitive dependencies amongst our list of functional dependencies. Finally, to assert that our relationship is in Boyce-Codd normal form, we require that all determinants of the relationship be candidate keys. This holds for our relationship since we only have one candidate key since two users can have the same password, age, and language but their uid must be unique since the uid is the primary key.

We went through this process for all of the relationships we derived from our E-R model. We would have included an example of normalizing a table which required us to split up tables but fortunately our initial design and translation ended up not requiring that.

5. PLAN MOVING FORWARD

The table 1 describes our progress so far as well as the plan moving forward. We have distributed the work among the team members to maximize the learning while also achieving the desired goals.

6. FALLBACK FOR OVER AMBITIOUSNESS

Our approach talks about implementing the application where we follow an MVC based approached where we interact with the database using JAVA based API's calls. The result obtained from these calls will then be displayed on the web UI. If we are not able to execute the planned idea, we have a fallback mechanism where we plan to display plain data in the form of JSON response object returned from the API calls instead of implementing the Javascript based UI.