Final Project Report for 2dv513 Database Theory
by Niall Thurrat (nt222fc)

1 Idea

Come up with an idea for your project. Describe what problem it solves, who the main user(s) will be, and why your idea is a good fit for them and the problem. Describe the main features that your application must have to be complete.

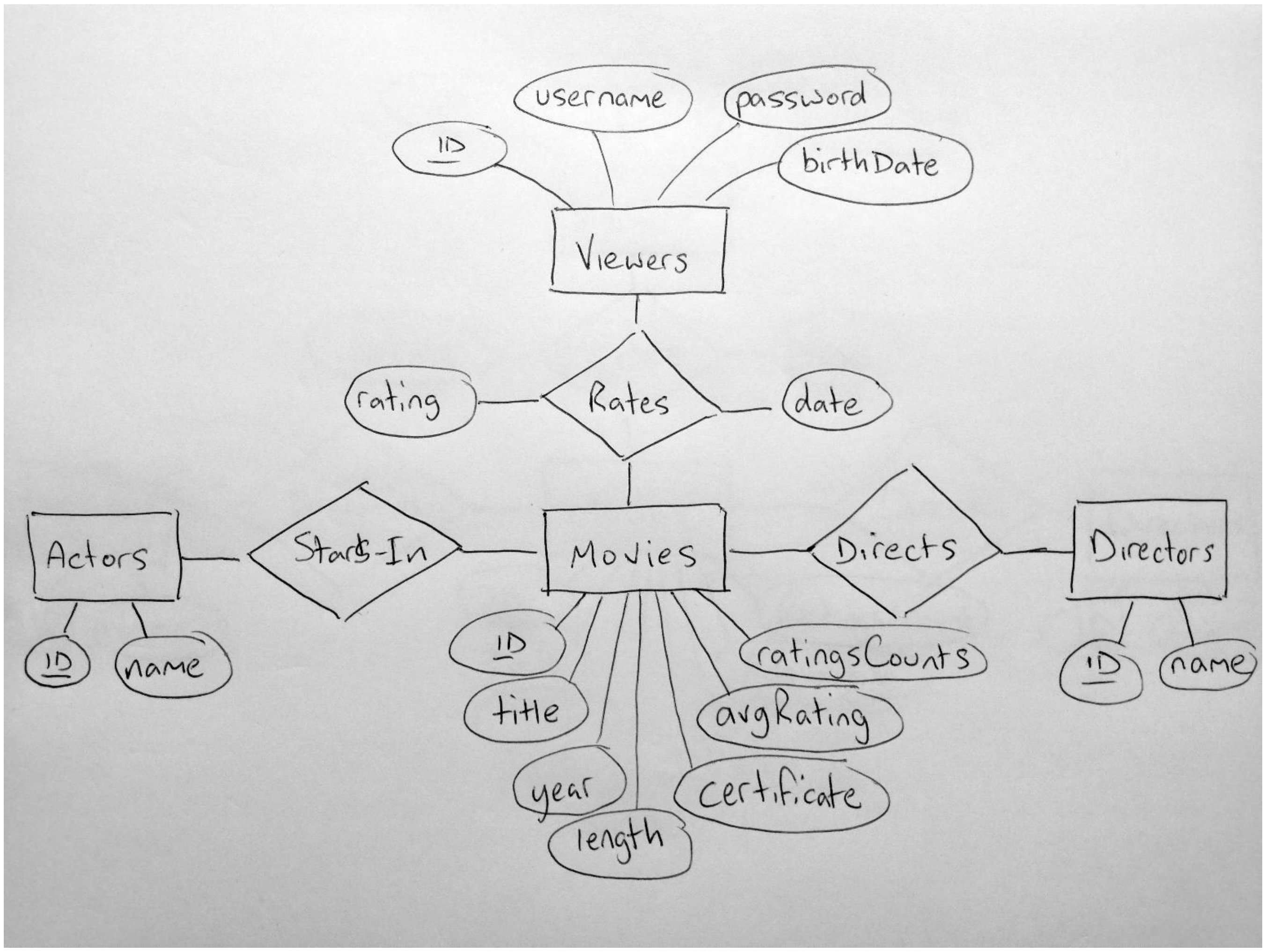
My project is to create a tool that allows users to rank movies and receive tips on other movies that they might be interested in watching based on actors and directors related to movies that they rate highly. This solves the common problem faced by many movie buffs: what movies should we watch next? The main users will be anyone interested in movies, from the novice movie viewer to the expert cinephile. My solution will be a good fit for them because it will use their top ranked movies to find other movies which share the same directors and actors, as well as showing top ranked movies (and associated people) by other viewers that use the tool. Main features must include:

- top 10 rated movies, actors and directors based on all viewers ratings
- viewer specific movie recommendations based on their own movie ratings
- registration and login of viewers
- views of the movie database based on a viewer's age group and movie certificates
- a search function for movies
- a 5-star rating function

I will obtain my data from IMDb.com.

2 Logical model

Design a data model for your project and present it as an E/R diagram. Make sure to include important attributes and relationships. Discuss and motivate your design.



My design has four entity sets, all with many-many relationships. Perhaps the most notable relationship is ‘Rates’ (Viewers – Rates - Movies) because this holds the attributes of each occasion a viewer rates a movie (the star ‘Rating’ and ‘Date’). These are crucial for calculating e.g. the movie rating averages that the tips are based on as well as recent movies with high ratings.

IDs have been used as keys in each entity set to make querying them easier.

Movies are given a ‘certificate’ attribute which will allow views of the movie table to be created for viewers who are underage and thus should not be given movie tips on movies they should not watch. I opted to add this attribute to the Movies entity set instead of having a separate set for movie certificates (e.g. Certificates(id, code, country)) to simplify my database and keep the scope of my project within the timeframe I have. I recognize that a movie in reality should have a one-many relationship with certificates as every country will provide its own certificate, but I am using the UK data that is provided to me in IMDb. The same could be said for the other attributes (year, title, length) which can be different in various countries, but I am using the UK data.

The ‘Movies’ set notably contains two attributes (‘avgRating’ and ‘ratingsCount’) that can be calculated using attributes from the ‘Rates’ relationship. However, I have decided to make these separate attributes in ‘Movies’ to reduce complexity in my queries and to avoid nesting queries which I believe could slow common queries in my planned application. Having these attributes already calculated against each movie will reduce the amount of tables that need to be queried in a lot of my app’s functions, e.g. selecting ‘Top 10 Movies (by Average Rating)’ will not need to query both ‘movies’ and ‘rates’.

3 Design In SQL

Translate your design to collections in SQL. Discuss and motivate how you translated entities and relationships.

viewers(ID, username, password, birth_date)
rates(viewerID, movieID, rating, date)
movies(ID, title, year, length, certificate, avg_rating, ratings_count)
stars_in(actorID, movieID)
actors(ID, actor_name)
directs(directorID, movieID)
directors(ID, director_name)

My database schema is a direct conversion of all entity sets and relationships to relations. This is done because all relationships are many-many. There are no weak entity sets, ‘Isa’ relationships/subclasses, or many-one relationships which require special consideration.

The name of ID attributes in relations which come from relationships (e.g. movieID) are changed for clarity, i.e. the tables contain multiple IDs and these must be renamed to ensure they are different. Name fields for actor and director are also expanded (e.g. actor_name) to enable easier referencing when their tables are joined together.

The ‘date’ attribute of the ‘rates’ relation refers to the last time the movie was rated. I am assuming that only one date is needed on the database. Therefore, even though I plan to allow viewers to change their rating, a history of this will not be held, i.e. the date will be over-written.

4 SQL Queries

Create five queries to your SQL design that are needed to implement the functionality of your application. You will probably need to create more than five queries to make your application functional, however we require some specific cases to be implemented and described in the assignment report. Focus on the more important queries and features of your application (i.e., there is no need to show how you insert documents in your various collections). Explain and motivate each query.

General guidelines for queries:

1. At least 2 queries should query data from more than one table, i.e., you should use at least two multirelation queries
2. You should make use of SQL JOIN
3. You should make use of Aggregation and/or Grouping
4. Create and use a View

Query 1 (multirelation query 1): Top 10 movies last week

```
SELECT title, AVG(rating) AS avg_rating, COUNT(rating) AS ratings_count
FROM rates
JOIN movies ON rates.movieID = movies.ID
WHERE date >= DATE(NOW()) - INTERVAL 7 DAY
GROUP BY movieID
ORDER BY avg_rating DESC
LIMIT 10;
```

This query gets the 10 movies with the highest average rating in the last week (last 7 days and today).
It fulfills my (main feature) goal to show the ‘top 10 rated movies based on all viewers ratings’ but is made even more relevant to users by focusing on recent ratings. It queries both the rates and movies relations. The rates table is queried first to get tuples with ratings and dates of all movies by movieID, while the movies table is joined to get the title of each movie for easier reading for app users. Unlike the ‘Top 10 Movies’ query which also shows on the home page, this query cannot simply use the movies relation (which has avg_rating and ratings_count pre-calculated) as the date of the rating is also necessary.

Query 2 (multirelation query 2): 10 movie tips based on directors of my favorite movies

```
SELECT movieID, title, year, avg_rating, director_name
FROM directs
JOIN movies ON directs.movieID = movies.ID
JOIN directors ON directs.directorID = directors.ID
WHERE directorID IN (
        SELECT * FROM (
                SELECT directorID
                FROM rates sq_rates
                JOIN directs ON sq_rates.movieID = directs.movieID
                WHERE sq_rates.viewerID = '10000'
                GROUP BY directorID
                ORDER BY ROUND( AVG(rating),1 ) DESC
                LIMIT 3
        ) AS sq
)
        AND movieID NOT IN (
                SELECT movieID
                FROM rates
                WHERE viewerID = '10000'
        )
GROUP BY movieID
```

```
ORDER BY avg_rating DESC
LIMIT 10;
```

This query gets the top 10 average rated movies that a viewer has not rated that have a director who is in the viewer's top 3 directors (by average movie rating). This query, along with a similar query based on the viewer's top 3 actors, is the apps most ambitious in terms of using the database to personalize each user's experience in the app and was created to fulfill my (main feature) goal to provide 'viewer specific movie recommendations based on their own movie ratings'. I use 2 sub-queries: the first gets the viewer's top 3 directors based on the viewer's average rating of their films; the second provides a list of movies that the viewer has rated that should not be included in the top 10 recommendations. The main query queries the directors relation first, as we must have all instances of movies being directed to be able to get only movies directed by the 3 directors returned by the first sub-query. The movies relation is joined to get titles for presentation, but also makes use of the avg_rating attribute to order and get the top ten movie tips. The directors relation is joined to get the director name for presentation to the app user. It is worth noting that my first sub-query has been wrapped in a separate select clause (*SELECT * FROM ([first sub-query]) AS sq*) as a work around for the fact that MySQL does not support *LIMIT* in sub-queries.

Query 3 (SQL JOIN): Gets movies by keywords (with viewer specific ratings)

```
SELECT ID, title, year, avg_rating AS average_rating, rating AS my_rating
FROM movies
LEFT JOIN (
        SELECT viewerID, movieID, rating
        FROM rates
        WHERE viewerID = 10000) AS my_rating
ON movies.ID = my_rating.movieID
WHERE title LIKE '%raging%'
        AND title LIKE '%bull%';
```

This query gets all movies with one or more specific strings in the title (also joins the ratings of a specific user). It fulfills my (main feature) goal to provide 'a search function for movies'. The query could be simplified by removing the join but I felt it was relevant to give the logged in user his/her own rating (if any) to show them if they have rated the movie before without them having to click the hyperlink to view the specific movie rating page. *LEFT JOIN* is used to add the user specific ratings to ensure that movies with no rating are included, i.e. movies that get a *null* rating attribute when the rates table is joined.

Query 4 (Aggregation and/or Grouping): Top 10 actors

```
SELECT actor_name, ROUND( AVG(avg_rating),1 ) AS act_avg_rating, SUM(ratings_count) AS ratings_count
FROM stars_in
JOIN movies ON stars_in.movieID = movies.ID
JOIN actors ON stars_in.actorID = actors.ID
GROUP BY actorID
ORDER BY act_avg_rating DESC
```

```
LIMIT 10;
```

This query gets the top 10 actors whose movies have the best average rating. It helps fulfills my (main feature) goal to show 'top 10 rated movies, actors and directors based on all viewers ratings'. The stars_in relation is queried first as this holds each instance that an actor has starred in a film. The joined movies relation provides a pre-calculated avg_rating for that specific movie. An average of the average rating of all movies for each actor can then be calculated by grouping the tuples by actorID and using the AVG() aggregate function.

Query 5 (Create and use a View): under_18_movies and under_15_movies views

```
CREATE VIEW under_18_movies
AS SELECT ID, title, year, length, certificate, avg_rating, ratings_count
FROM movies
WHERE certificate <> '18';

CREATE VIEW under_15_movies
AS SELECT ID, title, year, length, certificate, avg_rating, ratings_count
FROM movies
WHERE certificate NOT IN ('18', '15');
```

There are 2 views created in the database. These use the movie certificate attribute to serve views that exclude movies that should not be watched by viewers under a specific age (18 and 15). This is possible as users are required to enter a birth_date when they register. Registration and login are required to access the 'My page' and 'Rate movies' sections. This means that tips will not be given to users above their own age limit, they will not see these movies in their search results, and they will not be able to rate them.

5 Implementation

Write a program that implements your Idea in Task 1 with the design and queries from Task 2-4. You are of course allowed to introduce more queries.

The source code of my project can be found at https://github.com/niall-thurrat/movie_guru. The README file provides info on prerequisites and instructions on how to use the application locally.

6 Supplemental Video

Make a video (at most 10 minutes) demonstrating how your implementation works. It should show how it runs your queries and the results they produce (focus on guideline queries from Task4). You should upload the video somewhere (ex. YouTube or Vimeo), where it is accessible to us and reference it in the project report.

I have recorded a demo video of the application which can be found on YouTube at https://youtu.be/09iThXvfApU. The demo file is also in the project folder (DEMO-movie_guru-20200830_012213.mp4).