Project Team 6 - Popular Movie Database

Team Members: Grant Thornton and Sullivan Crouse

### Project Overview

Our project is creating a database out of the csv files sourced from this kaggle link: <https://www.kaggle.com/datasets/whenamancodes/popular-movies-datasets-9000-movies>. This dataset has 4 csv files containing information about 9000 popular movies. Since there are only 9000 rows, our scope isn’t very large. Since we have 4 files, we will create 4 tables. The context of this is that there were 600 users who posted 100000 reviews of about 9000 movies. The csv files are neatly made already so that we can easily create tables to implement in a relational database. This is because these files already have everything in terms of ids. This means we don’t need to do any data cleaning for this dataset. As stated earlier, our goal for this project is to convert these files into a relational database for use. This will allow us to better relate each file to each other.

### Environment Setup

Our database will contain 3 tables. For our user interface we will create a view of our database that has our tables joined to each other so that each movie review is shown with the corresponding information about the movie. If time permits, we might create a dashboard that utilizes our database schema. If this is done, it will be created using python as there are python packages, such as pandas, that can read from SQL statements. This will also be how we input our data into our databases. We will be utilizing a for loop and the mysql-connector-python package to insert our data into our tables.

### High Level Requirements

Have each of our csv files as a table in our database.

### Roles

Sullivan Crouse - Database implementation. Will work to create the database based off the database schema created by Grant Thornton. If a dashboard is created, will work to implement it with our database.

[Grant Thornton](mailto:jthornt9@uncc.edu) - Database strategy. Lead strategy development for the project. This will include the creation of the database schema, user interface requirements, and potentially the development of the dashboard requirements.

### Story Descriptions

|  |  |
| --- | --- |
| **Sprint** | **Goal** |
| Sprint 0 | Project Conceptualizations |
| Sprint 1 | Database and table creation |
| Sprint 2 | Database population |
| Sprint 3 | Quality assurance and user interface |

### High Level Conceptual Designs

The following tables will be included in our database with the following primary and foreign keys.

* Movie with primary key ‘movieid’
* Ratings with primary unique combination of ( ‘userid’,‘movieid’)
* Tags with primary unique combination of (‘userid’,‘movieid’)

All these tables are related to each other using movieid.

# Team 6 Sprint 1

### Refined Requirements:

* We will be using 3 different tables for our database. Movie, Ratings, and Tags.
* Movie: MovieID, Title, Genre with primary key MovieID
* Ratings: UserID, MovieID, Rating, Timestamp with unique combination primary key (UserID, MovieID)
* Tags: UserID, MovieID, Tag, Timestamp with unique combination primary key (UserID, MovieID)

### Conceptual Design:

To understand how our database was to be set up, the following entity relationship diagram was created.

Diagram

Description automatically generated

This entity relationship diagram was used to help refine how to build the tables for the database.

### Logical Design:

After the conceptual design was finished, the logical design of the database was created. In total 3 tables were utilized. They are shown below.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | Movie | | MovieID (PK) | | Title | | Genres | | |  | | --- | | Ratings | | UserID (Partial PK) | | MovieID (Partial PK) (FK) | | Rating | | Timestamp | | |  | | --- | | Tags | | UserID (Partial PK) | | MovieID (Partial PK) (FK) | | Tag (Partial PK) | | Timestamp | |

Because movieID was only unique in the movie table, it is the only single primary key in the entire schema. UserID couldn’t be used as a primary key by itself as UserID isn’t unique in any table. A user could have multiple ratings of just as there could be multiple movies rated. Because of this it was decided that the primary key of the ratings table be the combination of UserID and MovieID. MovieID was also referenced as a foreign key from the movie table since that is the best way to relate the two. Tags has a similar thing going on with it as it has multiple tags from the same user and movie. Because of this the primary key for tags was decided to be the 3-way unique combination of UserID, MovieID, and tag. To see how these tables look in action a select all query was run for each table. The results are below.

Movie:

Graphical user interface, text, application

Description automatically generated

Ratings:

Table

Description automatically generated

Tags:

Table

Description automatically generated

### Implementation and Deployment of Database:

After the conceptual and logical design of the database, the database was then created. A create database SQL statement was first used to create a database called movie. From there, a python script was used to both create and populate the tables with data. The python script and data used can be found in the attached zip file. The python script connects to a MYSQL database using a module called mysql.connector. The package can then allow you to write SQL code in strings in python to execute in a database. When looking for the SQL statements looking for lines of code that say mycursor.excecute.(). Running the script will populate the database specified with the movie table that has roughly 10,000 rows, ratings table with 100,000 rows, and tags table with 3600 rows.

### Key Queries:

Three key queries were created alongside the database population script. The first query finds the average rating and number of ratings for each movie. The second query finds the average rating each user gives. The final query counts the number of tags each movie is given. These three queries, while simple, can tell a lot of information both about the user, and about the movie. These queries weren’t implemented into a GUI yet but will be utilized when building the dashboard.

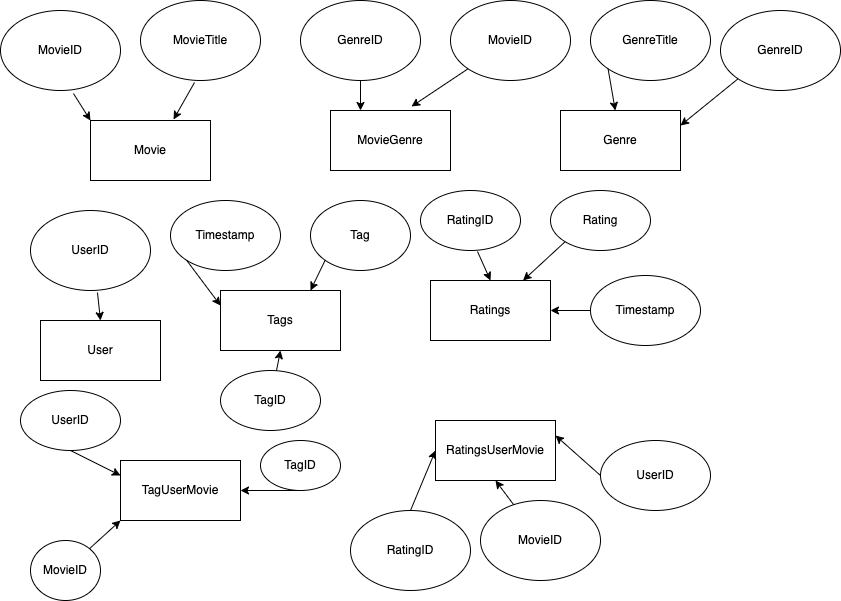
# Team Sprint 2

### Refined Requirements:

* We will be modifying the tables created in Sprint 1 and putting them into 4th Normal Form by reducing dependencies within the tables themselves.
* To do this we will be creating new tables “TagUserMovie”, “RatingsUserMovie”, “User”, “MovieGenre”, “Genre”.
* Movie: MovieID, MovieTitle
* MovieGenre: MovieID, GenreID
* Genre: GenreID
* User: UserID
* Tags: TagID, Tag, Timestamp
* Ratings: RatingID, Rating, Timestamp
* TagUserMovie: TagID, MovieID, UserID
* RatingsUserMovie: RatingID, MovieID, UserID

### Conceptual Design:

To better understand our database, the updated entity relationship diagram was created.



### Logical Design:

After the conceptual design was finished, the logical design of the database was created. In total 8 tables were utilized. They are shown below.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | Movie | | MovieID (PK) | | MovieTitle | | |  | | --- | | MovieGenre | | MovieID (FK) | | GenreID (FK) | | |  | | --- | | Genre | | GenreID (PK) | | genre | |
| |  | | --- | | User | | UserID (PK) | | |  | | --- | | Tags | | TagID (PK) | | Tag | | taggedAt | | |  | | --- | | Ratings | | RatingID (PK) | | Rating | | ratedAt | |
| |  | | --- | | TagUserMovie | | TagID (FK) | | MovieID (FK) | | UserID (FK) | | |  | | --- | | RatingUserMovie | | RatingID (FK) | | MovieID (FK) | | UserID (FK) | |  |

### Normalization:

The following tables are in 4th normal form: Movie, Genre, User, Tags, and Rating. All of these tables have complete candidate keys that don’t have transitive dependencies. Additionally, all the determinants of these tables are candidate keys. MovieGenre, TagUserMovie, and RatingUserMovie are not in 4th normal form. All of these tables are in 1st normal form as the candidate (primary key) for these tables are the combination of all of the columns in them. This schema was chosen as it allowed for us to relate the tables together while still being able to have the other tables in higher normalized forms.

### Database Implementation and Key Queries:

The database dump can be found in the attached zip folder along with the key queries for this sprint. The key queries for this sprint are a revising of the past queries so that they work with the current schema. There are also some new queries that are able to be utilized with the new normalized tables.These all have to do with the genre field as before this sprint it was much harder to work with. The first new query gets the average rating of a genre along with the number of ratings for that genre. The second finds the count of movies that are in each genre. The final finds the number of tags in each genre.

# Team Sprint 3

### Refined Requirements:

* We will not be modifying the tables created in Sprint 2
* We will change our queries into views for better user experience
* No stored programs will be created
* Index usage will be added

### Conceptual Design:

To better understand our database, the updated entity relationship diagram was created.

Diagram

Description automatically generated

### Logical Design:

After the conceptual design was finished, the logical design of the database was created. In total 8 tables were utilized. They are shown below.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | Movie | | MovieID (PK) | | MovieTitle | | |  | | --- | | MovieGenre | | MovieID (FK) | | GenreID (FK) | | |  | | --- | | Genre | | GenreID (PK) | | genre | |
| |  | | --- | | User | | UserID (PK) | | |  | | --- | | Tags | | TagID (PK) | | Tag | | taggedAt | | |  | | --- | | Ratings | | RatingID (PK) | | Rating | | ratedAt | |
| |  | | --- | | TagUserMovie | | TagID (FK) | | MovieID (FK) | | UserID (FK) | | |  | | --- | | RatingUserMovie | | RatingID (FK) | | MovieID (FK) | | UserID (FK) | |  |

### Normalization:

The following tables are in 4th normal form: Movie, Genre, User, Tags, and Rating. All these tables have complete candidate keys that don’t have transitive dependencies. Additionally, all the determinants of these tables are candidate keys. MovieGenre, TagUserMovie, and RatingUserMovie are not in 4th normal form. All these tables are in 1st normal form as the candidate (primary key) for these tables are the combination of all the columns in them. This schema was chosen as it allowed us to relate the tables together while still being able to have the other tables in higher normalized forms.

### Indexes

When looking at how the indexes for our tables will be structured, we have a mix of indexes being clustered and unclustered. Clustered indexes sort and store the data rows in the table based on their key values. While unclustered indexes have a structure separate from the data rows.

Tables that include clustered indexes are Movie, Genre, User, and Ratings. These are considered clustered because they each have a unique primary key that stores them in only one order. Tables that have unclustered indexes are MovieGenre, and RatingUserMovie because each key value entry has a point to the data row that contains the key value through foreign keys.

Tables:

* Movie
  + This table will have clustered indexes because of the nature of the primary key “MovieID”.
* MovieGenre:
  + This table will have unclustered indexes because it does not contain any primary keys and the two rows are both foreign keys.
* Genre:
  + This table will have clustered indexes because it contains the primary key, “GenreID”
* User:
  + This table will have clustered indexes because it contains the primary key, “UserID”.
* Tags:
  + The Tags table will have clustered indexes because of the “TagID” primary key.
* Ratings:
  + The Ratings table will contain clustered indexes as well because of the “RatingID” primary key.
* TagUserMovie:
  + This table will have unclustered indexes because it does not contain a primary key and only has foreign keys.
* RatingUserMovie:
  + This table will have unclustered indexes because it only contains foreign keys.

### Database Implementation and Views:

The database dump can be found in the attached zip folder. During this spring, no key queries were created. Instead, our past queries were transformed into views so that our program will be able to more easily utilize the database. These views are included in the new database dump.

Text, letter

Description automatically generated

No stored procedures were made as there was no need for any for this program.