Module Code: CS1003 Matriculation: 230005839 Tutor: Fahrurrozi Rahman

CS1003 Practical2: Databases

Overview

The objective of this assignment was to create a set of programs which would create a database, populate the database with values read in from separate files as well as be able to query the database and return relevant information. Engaging with all aspects of databases the assignment requires critical design thinking and developing a robust set of programs. For the assignment SQLite was the chosen database, working with JDBC in java to accomplish this. Going from ER diagram to querying a database has required a vast number of skills to be implored and appropriate knowledge of the JDBC API to be demonstrated.

Design & Implementation

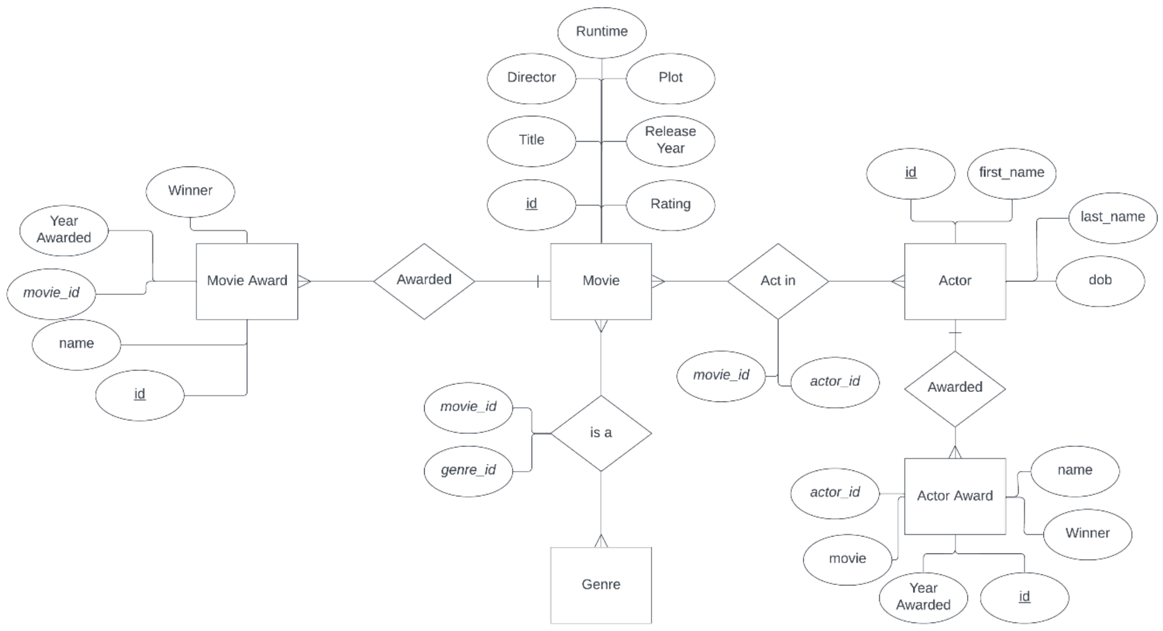

Figure 1: Final ER Diagram of DB

The first step to creating a database is a well-formed plan, in this case an Entity Relationship (ER) diagram showcases the multiple entities and the relationships that they will have. The program consists of five entities: Movie, Movie Award, Actor, Actor Award, Genre. These entities are then connected by various relationships, along with different restrictions on the number of relationships between each entity.

From this ER Diagram a database schema can be constructed to create the database. In the program the file createDB.sql contains the SQL commands to create all the necessary tables in the database. Seven tables are defined within the schema; The five entities are translated into tables, ensuring that appropriate types for each of the values are chosen. The "is a" and "Act in" relationships are also created as tables, movie_genre and movie_cast respectively. These relationships are transformed into tables as they are between a many to many relationship and so need to be further abstracted into a linking table. In all the tables the primary and foreign keys are explicitly stated, to ensure valid ids are used. An example of the data definition language can be seen below in the statement to create the movie_award table:

```
CREATE TABLE IF NOT EXISTS 'movie_award' (
id INTEGER PRIMARY KEY,
name TEXT,
movie_id INTEGER,
year_awarded TEXT,
winner INTEGER,
FOREIGN KEY(movie_id) REFERENCES movie(id)
);
```

The command above ensure that a duplicate version of the table isn't created. Commands similar are used to create all the other tables. At the beginning of the program all the tables are dopped if they exist, this ensures that a new version of the database is created every time the program is run.

```
DROP TABLE IF EXISTS movie;
DROP TABLE IF EXISTS actor;
DROP TABLE IF EXISTS movie_cast;
DROP TABLE IF EXISTS movie_award;
DROP TABLE IF EXISTS actor_award;
```

Now that a schema for the database exists, A java program to execute these commands was created. InitialiseDB.java contains two methods taken from previous example code: tryToAccessDB and checkDriverLoaded. This code was directly copied to ensure that access to the database would work. The createSchema method was edited from example code to incorporate reading the schema from a separate file. Lastly the validated method is original code; validateDB returns true if all the tables were created and exist in the database. This is achieved by searching through the meta data of the database, which includes the table names and other important information about the database. Running the program requires no command-line arguments, creating a fresh database each time it is run.

In order to add entries and fill the database PopulateDB.java was created. The information for entries comes from several CSV files found online. To make the data more efficient to process, some of the CSVs were saved as TSVs as splitting values based on a tab would lead to the correct entries. Information on the top 50 movies from IMDB, top 50 Actors, top 3 stars in movies and information on the Oscars Is found within these files.

Adding entries to each of the tables requires its own method with an equivalent format. Each method takes a connection to the database and the relevant fields for insertion. These fields are added to a preparedStatment and then are updated within the database. As an example, the InsertActor() method is given below:

```
public void InsertActor(Connection connection, String first_name,    String last_name, String dob)
throws SQLException{
   PreparedStatement statement;
   statement = connection.prepareStatement("INSERT INTO actor (first_name, last_name, dob)
VALUES ("
   + " ?,"
   + " ?,"
   + " ?" + ")");
   statement.setString(1, first_name);
   statement.setString(2, last_name);
   statement.setString(3, dob);

   statement.executeUpdate();
   statement.close();
}
```

The program first populates movies as this has the least dependencies in data. populateMovies() reads from the BestMovies.tsv and takes data from columns as input for inserting into the movie table. The file also contains all relevant genres of films, so the genres table is also updated during this method. The method checks if the genre already exists within the database, if not then it is inserted into the database. Both the ids for the movie and genre are recorded and inserted into the joining table, movie_genres. Overall, the method focuses on processing the text and inputting correct values into the database.

PopulateActors() then occurs, going through the same process as populateMovies(). Only information on the actors is in the HollywoodActors.csv so just the actors table is updated. Then to link the movies and actors together a third file IMDB_Movies7Actors.tsv is used to check the top three actors from 250 movies. populateCast() takes each of the actors listed in the file and links them with a movie. To further abstract this method the program uses linkMoviesActors(), this method searches for any matches of the title of the movie and name of the actor. If either of these values is not found then the program won't insert a value, if both the actor and movie are in the database already then, the link of cast is created, and the ids are put in the movie_cast table.

Finally populating the awards tables with populateAwards() reads from Oscars1927-2024.tsv is read to find all the relevant information for insertion. To determine if the award should be a movie award or actor award, the method check if the award title contains the word actor or actress. The Insert award methods both check to find the relevant id of either movie or actor as a foreign key in the table, done through a query to the relevant database.

All these methods are done one after each other in sequence to fill all the tables in the database. After the PopulateDB program is run the database is now fully set up and is ready to be queried.

Querying the database requires a separate program, QueryDB,java, to keep functions of classes separate and keep in line with good coding practices. The main method for QueryDB handles the command line arguments for the program as the user input. If the user inputs invalid values, then the programmed will use the inputError() method to print out a relevant message and exit the program. inputError() has an overloaded version of the method in which it takes an additional argument of a String to provide a more relevant error message about the usage of each of the options and additional command line arguments that they also require.

```
private void inputError(){
   System.err.println("USAGE: QueryQB # ");
   System.err.println("# Options are: ");
   System.err.println("1 - List the names of all the movies in the   database.");
   System.err.println("2 - List the names of the actors who perform in some specified movie.");
   System.err.println("3 - List the synopses of a movie with a specified actor in it and directed by
some particular director.");
   System.err.println("4 - List the directors of the movies that have a particular actor in them.");
   System.err.println("5 - ");
   System.err.println("6 - ");
   System.exit(0);
}
private void inputError(String errMessage){
System.err.println(errMessage);
... etc
```

To handle the six different queries that the user can perform on the database, a switch stamen was used to handle the different cases. Each case will handle additional input validation as they may require additional arguments for the query, then call the matching method for the query. Each query has its own method to keep the code clean. The four queries given are implemented correctly.

Listing the names of all the movies in the database requires a very simple query for the title from movie table. Then taking the resultSet returned and iterating over it, printing the titles from each row.

To list the names of the actors who perform in a specified movie, the user must also input a movie title that they wish to find. The query joins the actor and movie table together via the movie_cast table. The tables can join to each other as they contain relevant primary and foreign keys. Then the query specifies that the movie title should be equal to the user's input by using a preparedStatement. The method prints out all the actors in the database who are in the specified movie.

The next specified query was listing the synopses of a movie with a specified actor in it and a particular director. This query works on the same join statement as the previous query, as this joined table contains all the relevant information. The changes to the query come in changing the WHERE statement to the actor's name and director.

```
statement = connection.prepareStatement("SELECT movie.title, movie.plot FROM movie_cast "
+ "INNER JOIN movie ON movie_cast.movie_id = movie.id "
+ "INNER JOIN actor ON movie_cast.actor_id = actor.id "
+ "WHERE actor.first_name = ? AND actor.last_name = ? "
+ "AND movie.director = ?");
```

From this and the resultSet the method prints out the movie plots where the actor and director have worked together.

The final given query was to list the directors of the movies that have a particular actor in them. The query works the same as the last two, joining the movie, movie_cast and actor table. Searching this joined table for entries where the actor's name matches with user input. Then the methods prints the title and director from each of the movies the actor has been in.

The first own choice query finds all the awards an actor has been nominated or won. This query joins actor and actor_award to find the awards based on the actor's name. The method then prints out whether the actor won the award and for which movie and what year it was awarded as well. The SQL query can be seen below:

```
statement = connection.prepareStatement("SELECT * FROM actor_award "
+ "INNER JOIN actor ON actor_award.actor_id = actor.id "
+ "WHERE actor.first_name = ? AND actor.last_name = ?");
```

The second query finds the genres of films that won awards during a specified year. To get from the movie_award table to the genre table only two joins are necessary. The movie_award table contains a foreign key to a movie, the movie_genre table also contains the same foreign key, this results in the two tables to be joined on the same foreign key. Then the movie_genre table is joined to the genre table to retrieve the names of the genres, as movie_genre only holds a genre_id and not the name of the genre itself. It is also important to specify the year that the movie_award occurred in and that the record did win the award and wasn't just nominated. The SQL query is shown below:

```
statement = connection.prepareStatement("SELECT * FROM movie_award " + "INNER JOIN
movie_genre ON movie_award.movie_id = movie_genre.movie_id "
+ "INNER JOIN genre ON movie_genre.genre_id = genre.id "
+ "WHERE movie_award.year_awarded = ? AND movie_award.winner = TRUE ");
```

Now that the records have been returned the method goes onto print the genres. Ensuring that each genre is printed only once, the genres that have one are first put into a hashSet to store only unique values. The values from the hashSet are then printed out, ensuring that if one movie one multiple awards it's genres wouldn't be repeatedly printed.

Overall the entire project aims to achieve good programming practice. Each class focuses on one main function such as creating the database, populating the database and querying the database. Within each of these classes the main method is used primarily to handle user input and to start the functions. Methods are abstracted to create readable code and to ensure methods perform a single function.

Testing

Name of Test	Result	Name of Test	Result
Initialisation Tests		**PopulateDB Tests**	
initInitialiseDBTest	PASS	insertBradPittTest	PASS
initPopulateDBTest	PASS	insertBradPittTest2	PASS
initQueryDBTest	PASS	insertTitanticTest	PASS
		insertTitanticTest2	PASS
InitialiseDB Tests		insertRomanceTest	PASS
InitDBOKTest	PASS	insertDramaTest	PASS
validDBTest	PASS	insertMovieAwardTest	PASS
badSchemaTest	PASS	insertBadMovieAwardTest	PASS
		insertActorAwardTest	PASS
QueryDB Tests		insertBadActorAwardTest	PASS
case1Test	PASS	insertMovieGenreTest	PASS
movieNamesTest	PASS	insertBadMovieGenreTest	PASS
case2Test	PASS	insertCastMemberTest	PASS
fightClubCastTest	PASS	insertBadCastMemberTest	PASS
ryanCastTest	PASS	populateMoviesTest	PASS
case3Test	PASS	populateMoviesTest2	PASS
forrestGumpTest	PASS	populateActorsTest	PASS
theShiningTest	PASS	populateActorsTest2	PASS
case4Test	PASS	populateCastTest	PASS
christianBaleTest	PASS	populateAwardsMovieTest	PASS
mattDamonTest	PASS	populateAwardsActorTest	PASS
case5Test	PASS		
bradPittAwardsTest	PASS		
bradPittAwardsTest2	PASS		
mattDamonAwardsTest	PASS		
case6Test	PASS		
Genres2009Test	PASS		
Genres2009Test2	PASS		

Examples

To easily run all the program in sequence, the class Movies runs everything together. There is a variable in the class called query, which is the command line arguments for the QueryDB class. This variable is edited to test show different features.

Query option: 2, “Saving Private Ryan”

```
OK
Cast:
Tom Hanks
Matt Damon
```

Query option: 3, “Tom Hanks” and “Robert Zemeckis”

```
OK
Descriptions:
Forrest Gump: The history of the United States from the 1950s to the
'70s unfolds from the perspective of an Alabama man with an IQ of
75, who yearns to be reunited with his childhood sweetheart.
```

Query option: 4, “Brad Pitt”

```
OK
Movie: Director
Fight Club: David Fincher
```

Query option: 5, “Angelina Jolie”

```
OK
Awards:
Won for ACTRESS IN A SUPPORTING ROLE in "Girl, Interrupted" during
the 2000 Oscar's
Nominated for ACTRESS IN A LEADING ROLE in "Changeling" during the
2009 Oscar's
```

Query option: 6, 2003

```
OK
Winning Genres:
Drama
Music
```

Evaluation

The assignment had 6 requirements: Create and ER diagram, create a Relational Schema, create a Java program to create the Relational Schema, create a Java program to populate the database, create a Java program to query the database and write the project report. Figure 1 shows my ER diagram and fulfils the first criterion The Figure 1 demonstrates all the necessary Fields and entities mentioned within the specification. From this the relational schema was developed, which can be found in src/createDB.sql as a file containing DDL, completing the second criterion. Initialising the database from the relational schema also proved successful as seen in testing. The program will also output the desired “OK” if initialisation occurs as expected. The next criterion required the database to be populated, using the PopulateDB() class. The class successfully reads all the data from local files and inserts it into relevant tables in the database. Again, through thorough testing on the insertion and populating of each table, the program successfully populates the database with entities and relating tables. Finally, the database should be queried according to the four given queries and two own choices. Each query can be accessed via a different command line argument with the QueryDB() class, accurately printing out usage instructions if the command line arguments are invalid. All queries return expected results in easily readable formats for the user. As shown through testing and evaluation the program successfully fills all requirements set out in the specification.

Conclusions

The database design is easily understandable and allows for atomic data and easily traceable relationships between tables. PopulateDB() is the clearest class, as there are many examples of abstraction which makes the overall program more readable. However, this class was the one which required the most reworking and time. Finding an existing csv/data which contains all the fields that were required was impossible, settling for several files was the best option. The files downloaded made the most sense as they resulted in the least number of files to be downloaded. Working with the specific files caused several parts of the program to be less adaptable, with column indexes hard coded. Ideally the code would be more adaptable to a variety of files and layouts of data. This could be achieved by reading a header column and adapting to the specific layout of each file. In addition to PopulateDB(), QueryDB() was also an interesting class to develop. The general structure of each method is very similar, with a preparedStatement followed by retrieving the relevant data from the resultSet. The similarity of methods potentially may lead to further abstraction being possible, only further increasing the readability of the code. Further extending the project may lead to more efficiently populating the database by using streams, becoming necessary if larger files were used to populate the database. An improved design on the award tables by adding an award show column would allow for more shows to be read in, not only the Oscars. Overall, the implementation of the project was successful and demonstrated a high level of database understanding.