Overview

We were tasked with designing and creating a relational database to hold movie information. This included creating an ER-Diagram, a SQLite schema, as well as as a Java application to interface with the Database for adding to and querying the database.

Overall, I was very pleased with the design of my database, and the implementation worked to specification in an object-orientated, and somewhat efficient way using JDBC and ORMLite.

I gained valuable experience in mapping more complex relational designs in a Java application and adding information. Also, my application can be seen as a clear example of why databases are useful, as the raw data files which are parsed to create the database are much more difficult to understand than the final database, as elements have some complex relationships.

I apologize that this is incomplete, I had difficulties completing this practical in the time given. I hope this displays competency of the subject matter and I have attempted to complete all areas equally, as much as possible.

Design & Implementation

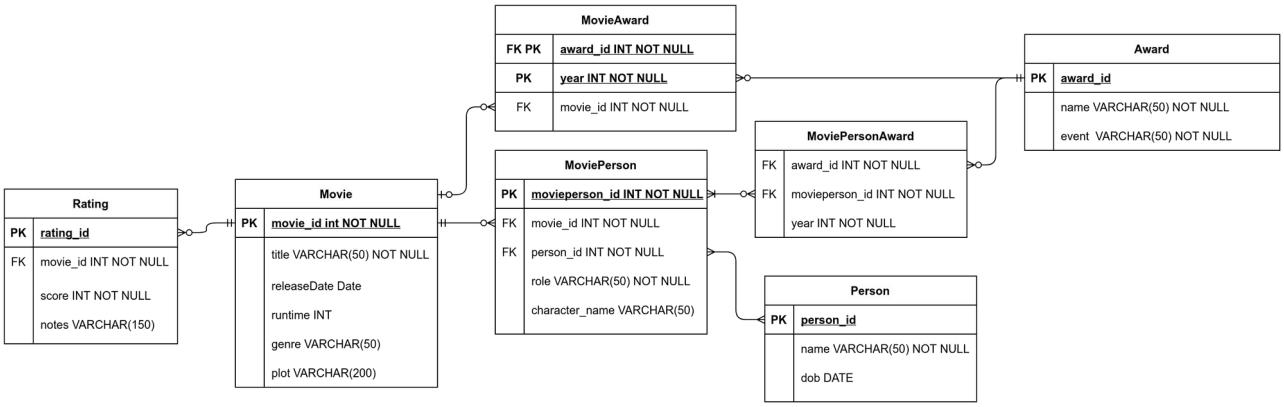

Figure 1: Final ER-Diagram created for the design of the database

To begin, I created an ER-Diagram with the information from the specification (Figure 1). This includes the first major database design decision of creating an intermediate table MoviePerson. This allows multiple actors to work on a movie and each actor to be able to work on multiple movies. It also allows other staff members to be added easily due to the role attribute. This design was chosen as adding new staff with different roles is much easier than adding a new column to the Movie entity for each new role, as well as allowing multiple directors, writers, etc. Thus, reducing the potential need to change the database design in the future.

The next decision I made was in adding in the two intermediate MoviePersonAward, and MovieAward entities. The reason for having these at all, instead of a sole Award entity was because I wanted to keep track of who won the same type of a award across the years.

This design also allows multiple MoviePersons to win the same Award in the same year, allowing group awards, as there are no primary keys and so the entire tuple must be unique. This is different for the MovieAward, as the year and award_id together make the primary keys and so multiple movies cannot win the same award in the same year.

My reasoning for having the CSV files and the way PopulateDB potentially drops data with small mistakes in an entry is with the focus on the scale of the data. Once data is in the database, the CSV files are not really needed in future if you don't re-initialise it and clear it and so the format of the CSV input files is less important. The worst outcome skipping over these entries which have mistakes could have is much of the data gets missed out. The mistakes can then be moved to a new CSV file, fixed, and added separately.

Examples & Testing

```
[george@thinker ~/Education/st-andrews/CS1003/Practicals/P3/CS1003-P3]$ java InitialiseDB
OK
[george@thinker ~/Education/st-andrews/CS1003/Practicals/P3/CS1003-P3]$ sqlite3 data/P3Datab
ase.db
SQLite version 3.35.2 2021-03-17 19:07:21
Enter ".help" for usage hints.
sqlite> .schema
CREATE TABLE IF NOT EXISTS 'Movie' (    'movie_id' INTEGER NOT NULL PRIMARY KEY AUTOINCREMEN
T,      'title' VARCHAR(50) NOT NULL,   'releaseDate' INTEGER,  'runtime' INTEGER,      'gen
re' VARCHAR(50),        'plot' VARCHAR(200));
CREATE TABLE sqlite_sequence(name,seq);
CREATE TABLE Review (   'rating_id' INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT, 'movie_id' I
NTEGER NOT NULL,        'rating' INTEGER NOT NULL,      'notes' VARCHAR(150),   FOREIGN KEY
('movie_id') REFERENCES Movie('movie_id'));
CREATE TABLE Person(    'person_id' INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT, 'name' VARCH
AR(50) NOT NULL,        'dob' DATE);
CREATE TABLE MoviePerson(       'movieperson_id' INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
'movie_id' INTEGER NOT NULL,    'person_id' INTEGER NOT NULL,   'role' VARCHAR(50) NOT NULL,
'character_name' VARCHAR(50),   FOREIGN KEY ('movie_id') REFERENCES Movie('movie_id'),  FORE
IGN KEY ('person_id') REFERENCES Person('person_id'));
CREATE TABLE Award(     'award_id' INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,  'name' VARCH
AR(50) NOT NULL,        'event' VARCHAR(50) NOT NULL);
CREATE TABLE MovieAward(        'award_id' INTEGER NOT NULL,    'movie_id' INTEGER NOT NULL,
'year' INTEGER NOT NULL,        FOREIGN KEY ('award_id') REFERENCES Award('award_id'),  FORE
IGN KEY ('movie_id') REFERENCES Movie('movie_id'),      PRIMARY KEY (award_id, year));
CREATE TABLE MoviePersonAward(  'award_id' INTEGER NOT NULL,    'movieperson_id' INTEGER NOT
 NULL,  'year' INTEGER NOT NULL,        FOREIGN KEY ('award_id') REFERENCES Award('award_id'
),      FOREIGN KEY ('movieperson_id') REFERENCES MoviePerson('movieperson_id'));
sqlite>
```

Figure 2: The initialisation step and the corresponding database file created with it's schema. This works exactly the same whether the database file already existed or not.

```
[george@thinker ~/Education/st-andrews/CS1003/Practicals/P3/CS1003-P3]$ java PopulateDB
OK
[george@thinker ~/Education/st-andrews/CS1003/Practicals/P3/CS1003-P3]$ sqlite3 data/P3Datab
ase.db
SQLite version 3.35.2 2021-03-17 19:07:21
Enter ".help" for usage hints.
sqlite> SELECT * FROM Movie
   ...> ;
1|The Shawshank Redemption|1994|147|adventure|Two imprisoned men bond over a number of years
, finding solace and eventual redemption through acts of common decency.
2|Joker|2019|146|action|In Gotham City, mentally troubled comedian Arthur Fleck is disregard
ed and mistreated by society. He then embarks on a downward spiral of revolution and bloody
crime. This path brings him face-to-face with his alter-ego: the Joker.
3|The Godfather|1972|152|fantasy|An organized crime dynasty's aging patriarch transfers cont
rol of his clandestine empire to his reluctant son.
```

Figure 3: The result of the PopulateDB class being run, with a snippet of the data showing successful permanent storage in the database file.

```
[george@thinker ~/Education/st-andrews/CS1003/Practicals/P3/CS1003-P3]$ java QueryDB 1
["The Shawshank Redemption","Joker","The Godfather","The Godfather: Part II","The Departed",
"Schindler's List","The Lord of the Rings: The Return of the King","Pulp Fiction","The Good,
 the Bad and the Ugly","The Lord of the Rings: The Fellowship of the Ring","Fight Club","For
```

Figure 4: The beggining of the output from the first query, printing all of the movies from the database in a list

```
[george@thinker ~/Education/st-andrews/CS1003/Practicals/P3/CS1003-P3]$ java QueryDB 2 "The
Godfather: Part II"
[{"actors":["Al Pacino","Robert Duvall","Diane Keaton","Robert De Niro"]}]
```

Figure 5: An example of the second query successfully running, printing out all of the actors in the movie given by it's title. Each object in the root list is a movie which matches the title

```
[george@thinker ~/Education/st-andrews/CS1003/Practicals/P3/CS1003-P3]$ java QueryDB 2
Error: 1 / 2 arguments given
Usage: java queryDB 2 <movie-name>
[george@thinker ~/Education/st-andrews/CS1003/Practicals/P3/CS1003-P3]$ java QueryDB 3
Error: 1 / 3 arguments given
Usage: java queryDB 3 <actor-name> <director_name>
[george@thinker ~/Education/st-andrews/CS1003/Practicals/P3/CS1003-P3]$ java QueryDB 4
Error: 1 / 2 arguments given
Usage: java queryDB 4 <actor-name>
[george@thinker ~/Education/st-andrews/CS1003/Practicals/P3/CS1003-P3]$ java QueryDB 5
Error: 1 / 2 arguments given
Usage: java queryDB 5 <award-count>
[george@thinker ~/Education/st-andrews/CS1003/Practicals/P3/CS1003-P3]$ java QueryDB 6
Error: 1 / 2 arguments given
Usage: java queryDB 6 <year>
```

Figure 6: What happens when each query is ran without enough arguments. It prints out the specific arguments needed for that query.

```
[george@thinker ~/Education/st-andrews/CS1003/Practicals/P3/CS1003-P3]$ java PopulateDB
Invalid value type: Unparseable date: "null"
OK
```

Figure 7: This shows what happens when a row in the CSV files is invalid. Here the null value is invalid and so the row is skipped but all the other values are still added .

Evaluation

The requirements for this practical were to design and implement a database to keep track of movie related information and I am particularly pleased with the database design itself and the flexibility of the code I produced to have potential to be adapted into many different programs if desired due to the modular design of classes only having functionality they necessarily require, such as keeping the query functions in the QueryDB class instead of the Database class.

The design has potential for scaling up in terms of the data that it holds and takes into consideration the variability of data that could potentially come up. However, I believe I could have done a better job at getting the data from the CSV files to begin with.

I believe the queries could have been dealt with in a more cohesive way. However, they complete the tasks they are designed to complete. I am just not particularly pleased with the code structure which could be improved upon.

Conclusions

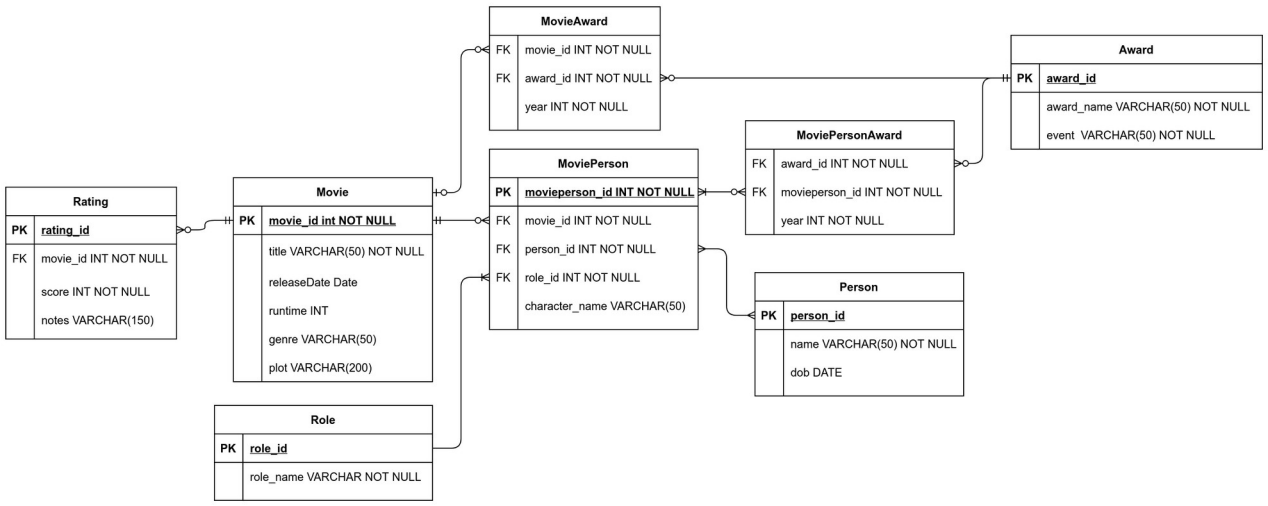

Figure 8: My original ER-Diagram, including the Role table and relationship to the MoviePerson entity

One difficulty I ran into whilst designing the database was whether or not to have a separate Role entity. My original ER-Diagram contained this table (Figure 8) which I quickly realised when creating the schema and basic Java classes was adding unnecessary complication to the model.

Given time to extend this application, it would be interesting to create a front-end application for this database, such as a website, which would display all of the information in an interactive way.