1. Overview:

For this project including database programming and data administration, a relational database for films, actors, directors, and awards has been designed, created, and manipulated using Java and SQLite. The primary requirements are creating an ER model, putting in place a relational schema, and using JDBC to connect Java to the database. Developing and initialising the database schema, adding structured data to tables with the help of an API, and utilising Java to run pre-written SQL queries are some of the main features that have been implemented. The program could be used to get movie titles, list actors in a certain film, and create complex queries based on the relationships between actors and directors, among other queries. To ensure database integrity and seamless program execution, strong error handling and testing techniques have been used. Extensive testing with JUnit has been carried out to validate input handling, query execution, and program performance against possible problems. While all core functionalities and requirements have been effectively implemented, further optimisations—such as using a better API could improve the amount of test data.

2. Design and Implementation:

The Design and Implementation has been divided in 2 sections; Section A will consist of the database design and Section B will consist of the software design.

Section A- Database Design

Please see the attached ER Diagram.pdf for the database table design.

ER Diagram Overview

The ER diagram represents a movie database system with entities for Movies, Actors, Directors, and Awards, along with their relationships. The design ensures data integrity by avoiding redundancy through proper normalisation.[1] [2] [3]

Key Entities:

1. Movies – Stores film details such as title, release date, running time, genre, plot, and ratings.

2. Actors– Contains actor-specific information like name and birthday.

3. Directors– Similar to actors, with attributes for name and birthday.

4. Awards– Defines awards with a name and category (e.g., "Oscar").

[1] https://www.datacamp.com/tutorial/normalization-in-sql
[2] https://www.freecodecamp.org/news/database-normalization-1nf-2nf-3nf-table-examples/
[3] https://medium.com/@ndleah/a-brief-guide-to-database-normalization-5ac59f093161

Relationships:

- Actors ↔ Movies: An actor can star in multiple movies, and a movie can feature multiple actors. This is resolved using the JOIN table Movie_Actors

- Directors ↔ Movies: A director can work on multiple films, and a movie may have multiple directors (JOIN table Movie_Director).

- Awards ↔ Movies and Awards ↔ Actors/Directors: Awards can be associated with movies or individuals. JOIN tables Movie_Awards, Actor_Awards, Director_Awards manage these relationships.

After designing the ER Diagram but before writing the full DDL script, I created preliminary tables to define the data types and constraints for each field. (Attached in the Appendix A.)

Relational Schema

The schema translates the ER diagram into SQL tables with primary keys (underlined) and foreign keys (represented with an *) to enforce referential integrity:

- Movies (movie_id, title, release_date, running_time, genre, plot, ratings)

- Actors (actor_id, name, birthday)

- Directors (director_id, name, birthday)

- Awards (award_id, name, category)

- JOIN Tables:

  - Movie_Actors (movie_actor_id, movie_id*, actor_id*)

  - Movie_Director (movie_director_id, movie_id*, director_id*)

  - Movie_Awards (movie_award_id, movie_id*, award_id*)

  - Actor_Awards (actor_award_id, actor_id*, award_id*)

  - Director_Awards (director_award_id, director_id*, award_id*)

Design Justification

- Normalisation: The schema avoids redundancy (e.g., awards are stored once and linked via IDs).

-Flexibility: JOIN tables support complex relationships (e.g., a movie winning multiple awards).

- Scalability: New attributes (e.g., role for movie_actor) can be added without restructuring.

Query Designs for the queries in QueryDB.java (requirement 5)

Query 1: List the titles of all the movies in the database.

Query Component	Details
Tables Needed	Movies
Columns to Select	title
Filter Conditions	None (all movies)
Joins	Not needed

Query 2: List the names of the actors who perform in some specified movie.

Query Component	Details
Tables Needed	Movies, Actors, Movie_Actors
Columns to Select	Actors.name
Filter Conditions	Movies.title = '[specified_movie_title]'
Joins	Movie_Actors to Movies Movie_Actors to Actors on actor_id

Query 3: List the plots of movies with a specified actor in them and directed by some particular director.

Query Component	Details
Tables Needed	Movies, Actors, Directors, Movie_Actors, Movie_Director
Columns to Select	Movies.plot
Filter Conditions	Actors.name = '[specified_actor]' AND Directors.name = '[specified_director]'
Joins	Movie_Actors to Movies Movie_Actors to Actors on actor_id Movie_Director to Movies Movie_ Director to Director on director_id

Query 4: List the directors of the movies that have a particular actor in them.

Query Component	Details
Tables Needed	Directors, Movies, Actors, Movie_Actors, Movie_Director
Columns to Select	Directors.name
Filter Conditions	Actors.name = '[specified_actor]'
Joins	Movie_Actors to Movies Movie_Actors to Actors on actor_id Movie_Director to Movies Movie_ Director to Director on director_id

Query 5: List movies that have won an Oscar and have a rating between 7.0-9.0 and lists the number of Oscars won.

Query Component	Details
Tables Needed	Movies, Movie_Awards, Awards
Columns to Select	m.title, m.ratings, COUNT(a.award_id) AS oscar_count
Filter Conditions	a.name = 'Oscar' AND m.ratings BETWEEN 7.0 AND 9.0
Joins	Movie_Awards to Movies on movie_id Movie_Award to Awards on award_id
Grouping	GROUP BY m.movie_id, m.title, m.ratings
Sorting	ORDER BY m.ratings DESC

Query 6: List all actors who have 2 or more awards and have starred movies with ratings of 8 and above.

Query Component	Details
Tables Needed	Actors, Movie_Actors, Movies, Actor_Awards
Columns to Select	a.name AS actor_name
Filter Conditions	m.ratings > 8.0
Joins	Movie_Actors to Actors on actor_id Actor_Awards to Actors on actor_id Movies to Movie_Actors on movie_id
Grouping	GROUP BY a.actor_id, a.name
Having Condition	HAVING COUNT(DISTINCT aa.award_id) >= 2
Sorting	ORDER BY COUNT(DISTINCT ma.movie_id) DESC

Section B

2,B.1 InitialiseDB

The InitialiseDB class is responsible for creating and initialising a movie database using SQLite. It ensures that any existing database file is deleted, a new database file is created and executes the necessary Data Definition Language (DDL) statements to create the required tables. The class also verifies the successful creation of tables after execution.

◊ **main(String[] args)**

The purpose of the main method is to manage the database creation process and calls helper methods for execution and verification. First step of the main method is to load the JDBC driver. The use of a try catch exception has been made so that my program handles the error gracefully on situations when the JDBC driver is not found, it suggests recompiling with the jar file in the classpath. The next job which my main method has is to delete the existing database files if there is any. Then it creates a connection to an SQLite database using a DriverManager.[4] If the connection fails or is null, it sends an error message and exits. It then runs the executeDDL() method to execute all the DDL statements listed in the file (the method has been explained below). Lastly the main method verifies if all the tables are created or not using the verifyTable() method.

◊ **executeDDL(Connection connection, String ddlFile)**

The purpose of the executeDDL method is to read SQL DDL statements from a file and executes them to create tables. I make use of a BufferReader to break the file down into input strings. I used a BufferReader instead of a Scanner because it is more efficient for handling larger inputs due to its buffering capabilities.[5] Additionally, BufferReader reduces the number of I/O operations, improving performance. The use of a StringBuilder has been made to construct the SQL query and stored as a String which gets executed. While the DDL file is getting read, all extra white spaces are getting trimmed using the trim() method[6]. All my comments in my DDL are also getting ignored, and if the line ends with a semi colon, the SQL gets executed. If successful, the program prints out a success message but if unsuccessful the program prints out an error message.

◊ **verifyTables(Connection connection)**

The purpose of this method is to checks if all required tables exist in the database after executing the DDL. The method is a Boolean as in the main method if it holds true, it will print

[4] https://docs.oracle.com/javase/8/docs/api/java/sql/DriverManager.html
[5] https://www.geeksforgeeks.org/difference-between-scanner-and-bufferreader-class-in-java/
[6] https://www.w3schools.com/java/ref_string_trim.asp

a success message else failure message. The names of the expected tables are stored as an array for later comparison. The method retrieves metadata information from the database using the getMetaData()[7] method of the Connection object. The getTables() method queries the database metadata to check if the table exists. The parameters passed (null, null, table, null) allow checking in the default schema without filtering by type. If rs.next() returns false, the table does not exist, and an error message is printed. If all tables exist, the method returns true. Any errors encountered (e.g., connection issues) are caught, logged, and the method returns false.

2.B.2 PopulateDB

The PopulateDB class is responsible for populating an existing SQLite database with data from CSV files. It first checks if the database exists, offers to initialise it if missing, then clears existing data while respecting foreign key constraints. The class reads data from multiple CSV files corresponding to different tables (actors, movies, directors, awards, and their relationships), validates the data format, and inserts it into the appropriate database tables with error handling for various SQLite error conditions. It provides detailed feedback about the population process and handles data integrity issues.

- ◊ **main(String[] args)**

The main method in this Java program initialises and populates an SQLite database using CSV files. Firstly, database file existence check has been added at the start of the main method. If database doesn't exist, the program prints error message, prompt user to enter 0 to initialise or any other key to exit, if user enters 0, the initialiseDB.sh script gets executed using ProcessBuilder[8]. If initialisation succeeds, recursively call main to retry population. The script's output is made visible in the console using pb.inheritIO(). Proper error handling for script execution has been added to ensure the program runs with minimum error possibilities.

The CSV files and queries have been stored in arrays, simplifying the population process. Storing CSV files and corresponding SQL queries in arrays allows for a structured and scalable approach to database population. Instead of writing repetitive code for each table, we can use a loop to iterate through the arrays and process multiple files efficiently. This improves maintainability, making it easy to modify or extend the database schema by simply updating the arrays. The use of a try-with-resources block has been implemented to automatically close the database connection. SQLite does not enforce foreign key constraints by default; hence PRAGMA foreign_keys = ON; has been used and it ensures referential integrity. The for loop iterates over CSV files and corresponding SQL statements. Then the method calls populateTable(connection, csvFiles[i], insertSQLs[i]) method which reads the CSV data, executes the corresponding SQL insert statement and returns true if successful. A try catch has been implemented to catch any SQLException which interrupted my project and based on the ErrorCode, a meaningful message gets displayed, for example Error code 1 is SQLite_ERROR,

[7] https://docs.oracle.com/javase/8/docs/api/java/sql/ResultSetMetaData.html
[8] https://docs.oracle.com/javase/8/docs/api/java/lang/ProcessBuilder.html

Error code 2 is SQLite_READONLY.[9] When the table is repopulated, the auto-incrementors counters get reset using the SQL query keyword 'DELETE' and the data gets reuploaded with a for loop.

◊ **populateTable(Connection connection, String csvFile, String insertSQL)**

The populateTable method reads data from a CSV file and inserts it into a database table using a prepared SQL statement. It ensures that the CSV data is correctly formatted before inserting it into the corresponding table. A try-with-resources block is used to manage both the BufferedReader (for reading the CSV file) and the PreparedStatement (for executing the insert queries), ensuring they are closed properly. A while loop iterates through each line of the CSV file. The first line (the header) is skipped. if (line.trim().isEmpty()) continue; ignores blank lines in the csv file and has been used to prevent inserting empty records into the database. Fields such as genres usually contain commas if there is more than 1 genre, hence a regex expression (",(?=(?:[^\"]*\"[^\"]*\")*[^\"]*$)") has been used to make sure only commas outside quotes are treated a delimiter.[10] After splitting the line into values, the method trims any extra whitespace and removes the unnecessary quotes which were used in the csv file to differentiate between the commas. It then verifies that the number of columns matches the number of placeholders (?) in the SQL statement. If the count does not match, an error message is printed, and the row is skipped. For each valid row, the method sets parameters in the PreparedStatement and attempts to execute the INSERT query. If any error occurs while inserting a row, an error message is displayed, but the method continues processing the remaining data. Finally, the method returns true if at least one row was successfully inserted; otherwise, it returns false.

2.B.3 QueryDB

The QueryDB class is designed to query my database based on user input. The program executes different queries depending on the argument passed when running it.

◊ **main(String[] args)**

Firsly the main method ensures that a query number is provided as an argument, if not it displays usage instructions. It then extracts the query number and database filename. It uses a try-catch to automatically close the database connection. With the help of the switch statement,

[9] https://www.sqlite.org/rescode.html
[10] https://stackoverflow.com/questions/40770990/split-string-by-comma-but-ignore-commas-in-brackets-or-in-quotes

it determines which query function to call based on user input and validates that required parameters for that specific query are provided.

- ◊ **listAllMovies(Connection connection) throws SQLException**

This method retrieves and prints all movie titles from the Movies table.

- ◊ **listActorsInMovie(Connection connection, String movieTitle) throws SQLException**

This method lists all the actors in the specific movie stated by the user.

- ◊ **listPlotsForActorAndDirector(Connection connection, String actorName, String directorName) throws SQLException**

This method finds movies where a specific actor and director worked together and returns the movie plot.

- ◊ **listDirectorsForActor(Connection connection, String actorName) throws SQLException**

This method finds all directors who have worked with a given actor.

- ◊ **complexQuery1(Connection connection) throws SQLException {**

This method finds movies that have won an Oscar and have a rating between 7.0-9.0 and lists the number of Oscars won. I have made use of the GROUP BY clause[11] to count the number of Oscars per movie while grouping by movie details (title, ratings). in SQL queries as well as the ORDER BY[12] clause to order by ratings.

- ◊ **complexQuery2(Connection connection) throws SQLException**

This method retrieves actors who have 2 or more awards and have starred movies with ratings of 8 and above. I made use of the HAVING[13] COUNT[14] clause in SQL queries as itfilters actors to only those who won at least 2 distinct awards, after grouping.

[11] https://www.w3schools.com/sql/sql_groupby.asp
[12] https://www.w3schools.com/sql/sql_orderby.asp
[13] https://www.w3schools.com/sql/sql_having.asp
[14] https://www.w3schools.com/sql/sql_count.asp

2.B.4 OMDBDataFetcher

The OMDBDataFetcher class fetches movie data from the OMDb API, processes the data, and writes it to CSV files for further use.

◊ **main(String[] args)**

The OMDb API URL is defined with an API key for authentication. Queries are made using title-based search (&t=) refers to movie title. The List of movie titles are stored in an array movieTitles which will be fetched from the OMDb API. Using the BufferedWriter, all the CSV files will be written on with the specific data they are expected to store. The .write()[15] method has been used to write the headers for the csv file. Keeping separate CSV files ensures data normalisation, preventing redundancy in the database. The API, extracts title, release date, runtime, genre, plot, and ratings from JSON files, the convertDateFormat() method has been called, (defined below) this method standardises the release date format for SQLite Databases.

◊ **JSONObject fetchMovieData(String title)**

First the API URL is constructed, replacing spaces with the movie titles. Then the code opens a connection[16] and reads the JSON response which has the data required to fill the CSV files. This method handles API failures gracefully by returning null in case of an error.

◊ **convertDateFormat(String oldDate)**

This method converts release date format from "DD MMM YYYY" to YYYY-MM-DD, which is the standard format for SQLite.[17] This method handles parsing errors by returning the original date if conversion fails.

2.B.4 The Script files

Script files simplify code execution, as covered in CS1007. These files automate the compilation process for convenience. The script files have been created to make the compilation easier. The initialiseDB.sh verifies that the schema exists. The populated.sh verifies that the csv directory exists. The Query.sh checks and verifies the arguments provided. Instead of manually running export CLASSPATH, javac, and java commands in the terminal, you can simply execute the corresponding script (e.g., `./initialiseDB.sh`), which handles these steps automatically. This reduces manual effort and minimises errors.

[15] https://docs.oracle.com/javase/8/docs/api/java/io/Writer.html
[16] https://docs.oracle.com/javase/8/docs/api/java/net/URL.html
[17] https://hyperskill.org/learn/step/27151

3. Examples

```
vc60@code-1:.../vc60/Documents/CS1003/CS1003-P3 $ ./initialiseDB.sh
Compilation successful. Running InitialiseDB...
SQLite JDBC driver not found.
Please add sqlite-jdbc.jar to your classpath.
```

Image 3.1: Initialisation with no driver loaded

```
vc60@code-1:.../vc60/Documents/CS1003/CS1003-P3 $ ./initialiseDB.sh
Compilation successful. Running InitialiseDB...
New database file created: database.db
DDL statements executed successfully.
OK - Database initialized successfully
```

Image 3.2: Initialisation with database file absent

```
vc60@code-1:.../vc60/Documents/CS1003/CS1003-P3 $ ./initialiseDB.sh
Compilation successful. Running InitialiseDB...
New database file created: database.db
DDL statements executed successfully.
OK - Database initialized successfully
```

Image 3.3: Initialisation with database file present

```
vc60@code-1:.../vc60/Documents/CS1003/CS1003-P3 $ ./populateDB.sh
Compilation successful. Running PopulateDB...
Database file does not exist. Please run InitialiseDB.
Enter 0 if you want to initialise the database and retry or any other key to exit.
g
exited
```

```
vc60@code-1:.../vc60/Documents/CS1003/CS1003-P3 $ ./populateDB.sh
Compilation successful. Running PopulateDB...
Database file does not exist. Please run InitialiseDB.
Enter 0 if you want to initialise the database and retry or any other key to exit.
0
Compilation successful. Running InitialiseDB...
New database file created: database.db
DDL statements executed successfully.
OK - Database initialized successfully
Database initialized successfully. Retrying PopulateDB...
csvfiles/actors.csv uploaded successfully.
csvfiles/movies.csv uploaded successfully.
csvfiles/directors.csv uploaded successfully.
csvfiles/awards.csv uploaded successfully.
csvfiles/movie_actors.csv uploaded successfully.
csvfiles/movie_director.csv uploaded successfully.
csvfiles/movie_awards.csv uploaded successfully.
csvfiles/actor_awards.csv uploaded successfully.
csvfiles/director_awards.csv uploaded successfully.
Database populated successfully.
```

Image 3.4 and 3.5: Population with database file absent

```
vc60@code-1:.../vc60/Documents/CS1003/CS1003-P3 $ ./populateDB.sh
Compilation successful. Running PopulateDB...
csvfiles/actors.csv uploaded successfully.
csvfiles/movies.csv uploaded successfully.
csvfiles/directors.csv uploaded successfully.
csvfiles/awards.csv uploaded successfully.
csvfiles/movie_actors.csv uploaded successfully.
csvfiles/movie_director.csv uploaded successfully.
csvfiles/movie_awards.csv uploaded successfully.
csvfiles/actor_awards.csv uploaded successfully.
csvfiles/director_awards.csv uploaded successfully.
Database populated successfully.
```

Image 3.6: Population with database file present

```
vc60@code-1:.../vc60/Documents/CS1003/CS1003-P3 $ ./queryDB.sh
Compilation successful. Running QueryDB with arguments:
Usage: ./queryDB.sh <query_number> [additional_parameters]
```

Image 3.7: Running the query file without a query number

```
vc60@code-1:.../vc60/Documents/CS1003/CS1003-P3 $ ./populateDB.sh
Compilation successful. Running PopulateDB...
Clearing existing data...
Cleared table: Actor_Awards
Cleared table: Director_Awards
Cleared table: Movie_Awards
Cleared table: Movie_Actors
Cleared table: Movie_Director
Cleared table: Actors
Cleared table: Directors
Cleared table: Movies
Cleared table: Awards
csvfiles/actors.csv uploaded successfully.
csvfiles/movies.csv uploaded successfully.
csvfiles/directors.csv uploaded successfully.
csvfiles/awards.csv uploaded successfully.
csvfiles/movie_actors.csv uploaded successfully.
csvfiles/movie_director.csv uploaded successfully.
csvfiles/movie_awards.csv uploaded successfully.
csvfiles/actor_awards.csv uploaded successfully.
csvfiles/director_awards.csv uploaded successfully.
Database repopulated successfully.
```

Image 3.7: Repopulating data

```
vc60@code-1:.../vc60/Documents/CS1003/CS1003-P3 $ ./queryDB.sh
Compilation successful. Running QueryDB with arguments:
Usage: ./queryDB.sh <query_number> [additional_parameters]
```

Image 3.8: Running QueryDB with no query number

```
vc60@code-1:.../vc60/Documents/CS1003/CS1003-P3 $ ./queryDB.sh 1
Compilation successful. Running QueryDB with arguments: 1
List of all movies:
1. Avengers: Endgame
2. Black Panther
3. Fight Club
4. Forrest Gump
5. Gladiator
6. Inception
7. Interstellar
8. Jurassic Park
9. La La Land
10. Parasite
11. Pulp Fiction
12. The Dark Knight
13. The Godfather
14. The Lion King
15. The Matrix
16. The Shawshank Redemption
17. The Social Network
18. Titanic
19. Toy Story
20. Whiplash
```

Image 3.9: Running query 1

```
vc60@code-1:.../vc60/Documents/CS1003/CS1003-P3 $ ./queryDB.sh 2
Compilation successful. Running QueryDB with arguments: 2
Usage: ./queryDB.sh 2 <movie_title>
```

Image 3.10: Running query 2 with no movie_title parameter

```
vc60@code-1:.../vc60/Documents/CS1003/CS1003-P3 $ ./queryDB.sh 2 "Inception"
Compilation successful. Running QueryDB with arguments: 2 Inception
Actors in movie 'Inception':
Leonardo DiCaprio
Joseph Gordon-Levitt
```

Image 3.11: Running query 2

```
vc60@code-1:.../vc60/Documents/CS1003/CS1003-P3 $ ./queryDB.sh 3
Compilation successful. Running QueryDB with arguments: 3
Usage: ./queryDB.sh 3 <actor_name> <director_name>
```

Image 3.12: Running query 3 with no actor_name and director_name parameter

```
vc60@code-1:.../vc60/Documents/CS1003/CS1003-P3 $ ./queryDB.sh 3 "Christian Bale" "Christopher Nolan"
Compilation successful. Running QueryDB with arguments: 3 Christian Bale Christopher Nolan
Plots of movies with actor 'Christian Bale' and director 'Christopher Nolan':
When the menace known as the Joker wreaks havoc and chaos on the people of Gotham, Batman must accept one of the greatest
 psychological and physical tests of his ability to fight injustice.
```

Image 3.13: Running query 3

```
vc60@code-1:.../vc60/Documents/CS1003/CS1003-P3 $ ./queryDB.sh 4
Compilation successful. Running QueryDB with arguments: 4
Usage: ./queryDB.sh 4 <actor_name>
```

Image 3.14: Running query 4 with no actor_name parameter

```
vc60@code-1:.../vc60/Documents/CS1003/CS1003-P3 $ ./queryDB.sh 4 "Robert Downey Jr."
Compilation successful. Running QueryDB with arguments: 4 Robert Downey Jr.
Directors of movies with actor 'Robert Downey Jr.':
Anthony Russo
Joe Russo
```

Image 3.15: Running query 4

```
vc60@code-1:.../vc60/Documents/CS1003/CS1003-P3 $ ./queryDB.sh 5
Compilation successful. Running QueryDB with arguments: 5
Movies with ratings between 7 and 9 that have won Oscars:
1. The Dark Knight (Rating: 9.0) - 2 Oscars
2. Pulp Fiction (Rating: 8.9) - 2 Oscars
3. Inception (Rating: 8.8) - 1 Oscar
4. Fight Club (Rating: 8.8) - 1 Oscar
5. Forrest Gump (Rating: 8.8) - 2 Oscars
6. The Matrix (Rating: 8.7) - 1 Oscar
7. Interstellar (Rating: 8.6) - 1 Oscar
8. Parasite (Rating: 8.5) - 1 Oscar
```

Image 3.16: Running query 5

```
vc60@code-1:.../vc60/Documents/CS1003/CS1003-P3 $ ./queryDB.sh 6
Compilation successful. Running QueryDB with arguments: 6
Actors who won 2 or more awards and starred in >8.0 rated movies:
1. Tom Hanks
2. Morgan Freeman
3. Heath Ledger
```

Image 3.17: Running query 6

4. Testing

All testing has been completed with the implementation of JUnit. All the JUnit tests pass gracefully, evidence is attached in Appendix B. The file test.sh can be used for running the tests (./tests.sh).

What is being tested?	Name of the method	Preconditions	Expected outcome	Actual Outcome
InitialiseDBTest				
Database file creation	testDatabaseFileExists()	Database file does not exist	Database file should exist after initialisation	Pass
Table creation in database	testTableCreation()	Database is empty	'Actors' table should be created	Pass
Database cleanup and recreation	testDatabaseDeletion()	Database file exists	File should be deleted and recreated	Pass
Debug table listing	debugTables()	Database is initialised	Should print all table names without errors	Pass
PopulateDBTest				
Database connection	testDatabaseConnection()	Database is initialised	Connection should be established	Pass
Existence of required CSV files	testCSVFilesExist()	Database is initialised	All required CSV files should exist	Pass
Population of database tables	testTablePopulation()	Database schema exists	All tables should contain data after population	Pass
Foreign key constraints	testForeignKeyViolations()	Database is populated	Should prevent invalid foreign key inserts	Pass
Duplicate entry handling	testDuplicateEntries()	Database is initialised	Duplicates should be allowed unless UNIQUE constraint exists	Pass
Performance with large datasets	testPerformanceWithLargeCSV()	Database is initialised	Should handle 10,000 record inserts	Pass
Empty CSV file handling	testEmptyDataFiles()	Database is initialised	Should handle empty CSV files without errors	Pass
Malformed CSV file handling	testMalformedDataFiles()	Database is initialised	Should handle malformed CSV files gracefully	Pass

Data type mismatch handling	testDataTypesMismatch()	Database is initialised	Should reject invalid data types	Pass
Missing required fields handling	testMissingRequiredFields()	Database is initialised	Should reject records with missing required fields	Pass
QueryDBTest				
Basic movie query functionality	testListAllMovies()	Database is populated	Should return count of movies > 0	Pass
Actor-movie relationship query	testListActorsInMovie()	Database is populated	Should return count of actor-movie relationships > 0	Pass
Director-actor relationship query	testListDirectorsForActor()	Database is populated	Should return count of director-movie relationships > 0	Pass
Complex query 1 functionality	testComplexQuery1()	Database is populated	Should return count of movie awards > 0	Pass
Complex query 2 functionality	testComplexQuery2()	Database is populated	Should return count of actor awards > 0	Pass
Handling of queries with no results	testNoResultsQueries()	Database is populated	Should handle non-existent movie query gracefully	Pass

5. Evaluation

All the requirements for the CS1003 Practical 3 project have been successfully implemented in my program. Throughout the development process, I gained a deeper understanding of ER modelling, relational schema design, and database integration using JDBC. Implementing SQLite database operations in Java allowed me to enhance my skills in SQL scripting, prepared statements, and exception handling. Additionally, I successfully implemented a structured approach to querying the database, ensuring efficient data retrieval through optimised SQL queries. Error handling was incorporated to manage unexpected issues such as missing data files, incorrect input formats, and database connection failures.

Despite the successful implementation, there are areas that could be improved. The current functionality meets the project specifications, but additionally I used the OMDb API which did provide all the information about movies, actors and awards, but it did not provide information about the directors which required me to manually look for them. I could have improved by using a more efficient API rather that searching and adding all the directors manually.

Overall, I am satisfied with my submission, as it meets all the specified requirements and demonstrates a solid understanding of database integration in Java. While the solution is functional and efficient, future improvements could focus on performance optimisation.

6. Conclusion

I thoroughly enjoyed working on this project, particularly the challenge of integrating an API into my project to get movie data. Designing the database schema, writing SQL queries, and ensuring seamless interaction between the Java program and SQLite database were both engaging and rewarding. One of the most interesting aspects was optimising queries for efficient data retrieval while handling potential errors gracefully. I encountered challenges in managing database connections effectively, ensuring the correctness of query results, and integrating the API, but through persistent debugging and refinement, I was able to develop a functional solution. If I had more time, I would like to further optimise my dataset with more data and information using a better API.

Overall, this project has been an excellent learning experience, strengthening my skills in database design, SQL, and Java programming. I look forward to applying these skills to future projects and exploring more advanced database-driven applications.

7. References

[1] https://studres.cs.st-andrews.ac.uk/CS1003/

[2] https://studres.cs.st-andrews.ac.uk/CS1007/

[3] https://www.datacamp.com/tutorial/normalization-in-sql

[4] https://www.freecodecamp.org/news/database-normalization-1nf-2nf-3nf-table-examples/

[5] https://medium.com/@ndleah/a-brief-guide-to-database-normalization-5ac59f093161

[6] https://docs.oracle.com/javase/8/docs/api/java/sql/DriverManager.html

[7] https://www.geeksforgeeks.org/difference-between-scanner-and-bufferreader-class-in-java/

[8] https://www.w3schools.com/java/ref_string_trim.asp

[9] https://docs.oracle.com/javase/8/docs/api/java/sql/ResultSetMetaData.html

[10] https://docs.oracle.com/javase/8/docs/api/java/lang/ProcessBuilder.html

[11] https://www.sqlite.org/rescode.html

[12] https://stackoverflow.com/questions/40770990/split-string-by-comma-but-ignore-commas-in-brackets-or-in-quotes

[13] https://www.w3schools.com/sql/sql_groupby.asp

[14] https://www.w3schools.com/sql/sql_orderby.asp

[15] https://www.w3schools.com/sql/sql_having.asp

[16] https://www.w3schools.com/sql/sql_count.asp

[17] https://docs.oracle.com/javase/8/docs/api/java/io/Writer.html

[18] https://docs.oracle.com/javase/8/docs/api/java/net/URL.html

[19] https://hyperskill.org/learn/step/27151

8. Appendix

Appendix A

Movies		
Name	Type	Detail
movie_id	INTEGER	Primary Key, Autoinc
title	TEXT	Movie Title
release_date	DATE	Release Date
running_time	INTEGER	Duration in minutes
genre	TEXT	Movie Genre
plot	TEXT	Plot Summary
ratings	REAL	Rating (0.0-10.0)

Actors		
Name	Type	Detail
actor_id	INTEGER	Primary Key, Autoinc
name	TEXT	Actor Name
birthday	DATE	Date of Birth

Directors		
Name	Type	Detail
director_id	INTEGER	Primary Key, Autoinc
name	TEXT	Director Name
birthday	DATE	Date of Birth

Awards		
Name	Type	Detail
award_id	INTEGER	Primary Key, Autoinc
name	TEXT	Award Name
category	TEXT	Award Category

Movie_Actors		
Name	Type	Detail
movie_actor_id	INTEGER	Primary Key, Autoinc
movie_id	INTEGER	Foreign Key (Movies)
actor_id	INTEGER	Foreign Key (Actors)

Movie_Director		
Name	Type	Detail
movie_director_id	INTEGER	Primary Key, Autoinc
movie_id	INTEGER	Foreign Key (Movies)
director_id	INTEGER	Foreign Key (Directors)

Movie_Awards		
Name	Type	Detail
movie_award_id	INTEGER	Primary Key, Autoinc
movie_id	INTEGER	Foreign Key (Movies)
award_id	INTEGER	Foreign Key (Awards)

Actor_Awards		
Name	Type	Detail
actor_award_id	INTEGER	Primary Key, Autoinc
actor_id	INTEGER	Foreign Key (Actors)
award_id	INTEGER	Foreign Key (Awards)

Director_Awards		
Name	Type	Detail
director_award_id	INTEGER	Primary Key, Autoinc
actor_id	INTEGER	Foreign Key (Actors)
director_id	INTEGER	Foreign Key (Directors)

Appendix B