Film Database (FDb)

Brandon Alexander Burtchell

Department of Computer Science, Texas State University

CS 5332: Database Theory and Design

Dr. Shounak Roychowdhury

May 4, 2023

I. INTRODUCTION

The artform of film is one of relatively recent inception (barely more than a century old, which is extremely young compared to visual art, music, or literature), yet it has achieved mass appeal, intense artistic standards, and a diverse history of styles and genres. In the present, the proliferation of video streaming (YouTube, Netflix, etc.) as the primary way in which viewers watch films, has democratized the art form to allow filmmakers around the world to self-release their work, in addition to the traditional industry route of production and release. This has ballooned the amount of films released on a yearly, if not weekly or daily basis, meaning it is important to ensure art is not forgotten forever to the constant buzz. Creating a database application with the ability to store detailed information about films, as well as the careers of filmmakers, is essential to cataloging and archiving the art form for future generations.

Databases like this do exist for film and television. Most popular of which is IMDb (Internet Movie Database) [1], which has expanded into offering a movie streaming service of its own. Various alternatives exist, such as TMDB (The Movie Database) [2], which is community-built with an extensive API used by applications such as Letterboxd [3] to create a social-networking experience centered around logging, rating, reviewing, and sharing films.

The goal of my database implementation is not to necessarily revolutionize the ways in which a film database could be engineered and designed, but rather to use a complex problem such as this as an opportunity to put my knowledge of database design into practice. However, I did intentionally include certain entities in the database that are not usually found in common databases. Most notably, I include a "Franchise" entity that groups up films that are part of the same series or franchise (for example, films in the Marvel Cinematic Universe, the Fast and the Furious series, the Before Trilogy, etc.). By grouping films in this way, users can easily track their progress through watching a film series, rather than having to individually search for each individual film in the series.

The rest of the paper is structured as follows: Section II introduces the application from a high-level view. Section III describes the database design in detail, with an entity-relation (ER) diagram and examples of SQL code and valid queries. This section will also include performance results. Section IV describes the development process and discusses the challenges faced and my solutions. This section also outlines the opportunities for future work. Finally, Section V is the conclusion, and Section VI contains the full list of references mentioned throughout this report.

Source code of this project is viewable at https://github.com/dukeofjukes/cs5332-final.

II. APPLICATION

Since I was working on this project alone, all aspects of the application—including design, planning, SQL programming, and report/presentation preparation—were completed by myself. The next section logs the step by step implementation of the database application in its entirety.

The application, titled Film Database (FDb) is entirely developed in Oracle SQL Plus. Due to the time constraints of the project I did not attempt creating a working web application with a frontend. Doing so would be unfeasible in such a short time and would likely result in a worse backend product. So, all database accesses and queries are done through the Oracle SQL Plus shell. Section III-D contains examples of interesting/useful queries to the database that could, in theory, be called from a frontend interface in a fully implemented web application.

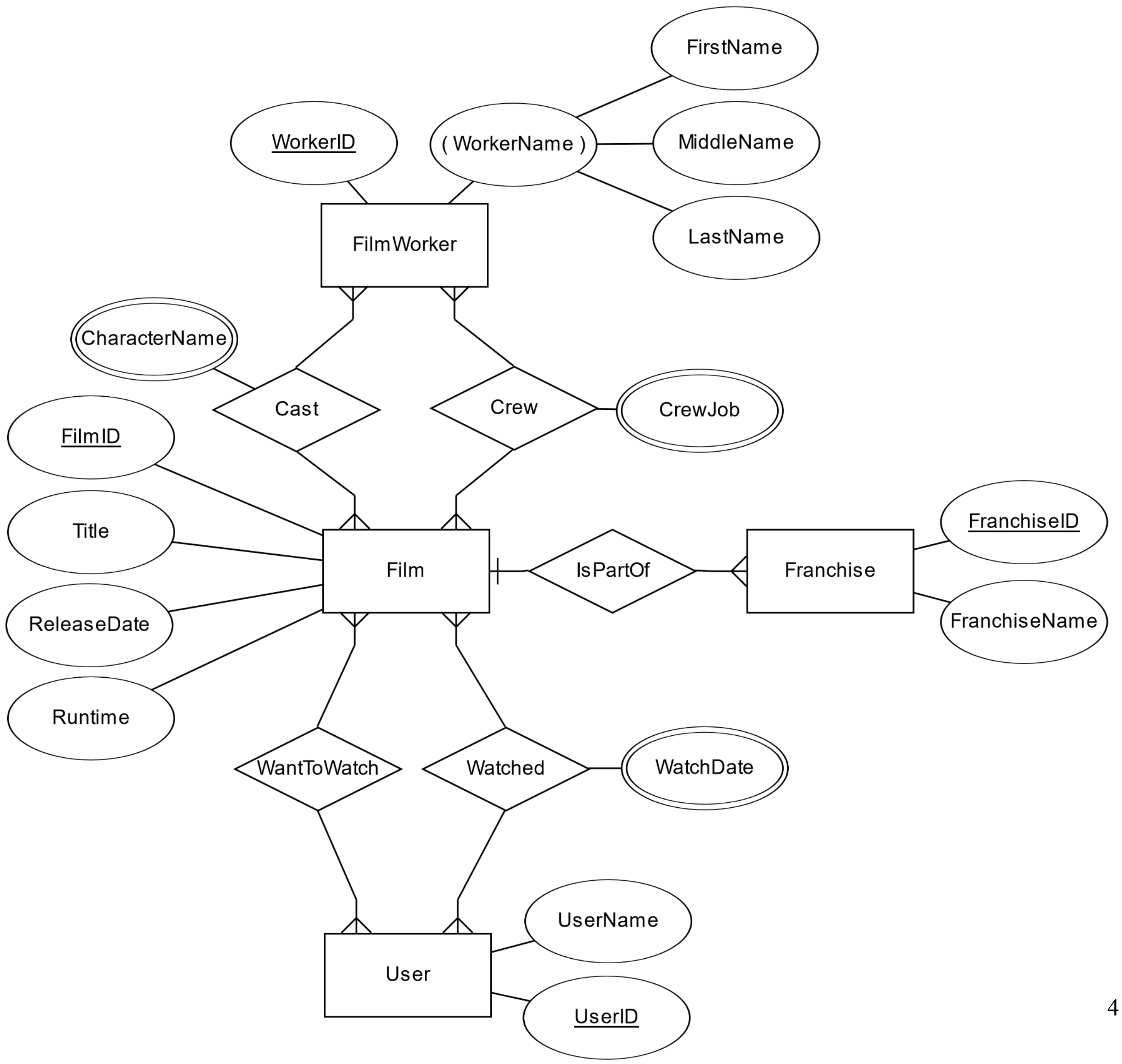
III. DATABASE IMPLEMENTATION

This section is split into 5 parts, weaving the development story of this database application. First, in part A, I describe the initial ER representation of the database in detail.

Then, in part B, I present the relational schema representation, and describe the process of translating from ER to relational schema. Next, part C shows examples of the actual SQL representation of the database, with code examples. Next, part D lists some useful example queries that could, in a full-scale application, be passed to a JavaScript web app to access information. In the case of this SQL application, these queries are provided as views. Lastly, part E provides some performance/efficiency analysis on this proof-of-concept database.

A. ER Representation

I first began with designing an entity-relation (ER) diagram to have a visual overview to guide my development. Below is the full ER diagram.



I first began by identifying entities that would be necessary to build the backbone of a film database, while keeping the scope within reason for a solo project. I also wanted to ensure each entity had enough attributes to allow for interesting and useful queries once I translated the ER database to a relational SQL schema. At the core, I identified a Film entity. A film has a unique ID number (arbitrarily assigned), a title, release date, and runtime. The inclusion of runtime and release data as attributes was to allow for films with the same title (i.e. remakes) to have other (likely) unique attributes, aside from the internal ID number. Since films are the core of the database, all other entities were defined with relationships with the Film entity.

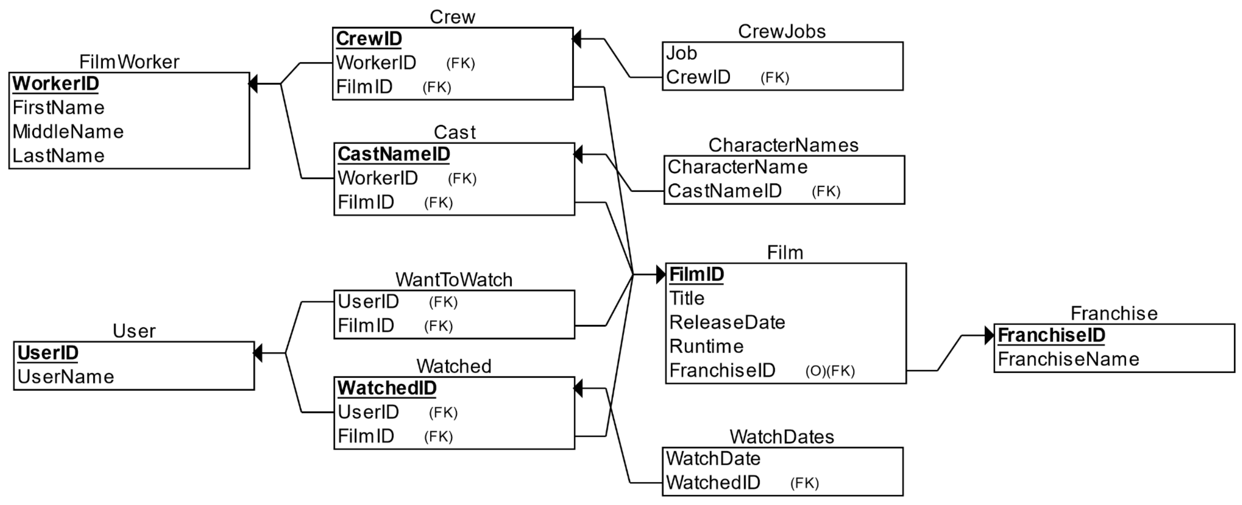
To handle users of the imaginary app, I defined a User attribute. For the sake of simplicity, a User only has a UserName and UserID. Users can have two possible relationships with a film: Watched or WantToWatch. These are N:M relationships. The Watched relationship has a multivalued attribute—WatchDate—to allow a user to log a film many times.

A FilmWorker entity is defined to represent any cast, crew, producers, directors, etc. of a film. This entity enables queries such as: “Which films are directed by Stanley Kubrick?”, or “How many films has Tom Cruise starred in?”. The type of film worker will be defined by their M:N relationship(s) to the film: Cast or Crew. The Cast relationship has a multivalued attribute—CharacterName—to store what character(s) a particular film worker played in a film. The reason it is multivalued is to allow cases in which an actor plays many roles (i.e. Mike Myers in the *Austin Powers* franchise). The Crew relationship also has a similar multivalued attribute—CrewJob—which stores which responsibilities a particular film worker had on a film. This is multivalued for a similar reason as the CharacterName, since a worker can have multiple “jobs” on a film (i.e. Quentin Tarantino has written, produced, and directed, most of his feature films).

Finally, a Franchise entity is defined to group together films that are part of the same series or franchise. For simplicity's sake, I make this a 1:N relationship (IsPartOf), where a film can only be part of one franchise, and a franchise contains many films. While there are films that are part of multiple franchises (i.e. *Spider-Man: No Way Home (2021)* belongs to both the *Spider-Man* series and the *Marvel Cinematic Universe*), these are outliers, and can simply be said to belong to their larger "parent" franchise (i.e. *Spider-Man: No Way Home (2021)* is part of the *Marvel Cinematic Universe* franchise).

B. ER to Relational Schema

After designing an ER model of the database, I translated it to a relational diagram to prepare it for representation in SQL. This procedure was achieved using the process outlined in Chapter 9 in the class textbook [4]. Below is the full diagram representing the resulting relational schema.



First, I mapped regular entities (Film, User, Franchise, FilmWorker) into their own entity relations. Each of these regular entities' attributes became attributes of their corresponding entity relation. Composite attributes (i.e. WorkerName) were simply broken down into their constituent attributes (FirstName, MiddleName, LastName). Since each of my entities were strong entities (no inheritance, etc.), I did not have to map any weak entity types.

Next, I mapped relationship types into relationship relations. Since my ER design did not include any 1:1 relationships, I did not have to handle them. The IsPartOf relationship, however, had to be mapped to the relational schema. This being a very simple relationship, I opted to take the foreign key approach, where the N-side entity (the Film, in this case) gets a foreign key referencing the primary key of the 1-side entity (the Franchise, in this case). This approach avoids the need of defining an extra relationship relation, leading to a more streamlined schema in the end.

Next, I mapped binary M:N relationship types. This included the Cast, Crew, Watched, and WantToWatch relationships. Each of these were mapped by creating a relationship relation, which include foreign keys referencing the participating entities. While in a theoretical relational schema, the primary key of these kinds of relation is usually the combination of the two foreign keys, I opted to also define a new primary key for each of the relationship relations, to facilitate an easier SQL implementation. This allowed for explicit references to these relations in the next step, where I map multivalued attributes.

To map multivalued attributes, I had to define new relations for each of them. In this specific case of my database's design, only the Cast, Crew, and Watched relationships contained multivalued attributes. So, each of these new multivalued relations contain a foreign key pointing to their corresponding relation, allowing multiple multivalued relations to exist for each relationship relation. They also contain an attribute that stores the value that the attribute was originally designed to contain (i.e. the WatchDate). This retains the design that was expressed by the original ER expression.

Overall, translation from ER to relational schema was a straightforward process that allowed me to create a top-level design and represent it in a way that facilitates easier modification and accessing via SQL.

C. SQL Representation

Representing the relational schema in SQL was a relatively straightforward process. The only new information that was needed was to decide on types for each attribute, as well as ensuring the proper primary key, foreign key, and null constraints were applied. The code below demonstrates how all relations were declared.

```
-- entity relations

create table Franchise (
  FranchiseID int primary key,
  FranchiseName varchar(40) not null
);

create table FilmWorker (
  WorkerID int primary key,
  FirstName varchar(20) not null,
  LastName varchar(20) not null
);

create table AppUser (
  UserID int primary key,
  UserName varchar(20) not null
);

create table Film (
  FilmID int primary key,
  Title varchar(40) not null,
  ReleaseYear int not null,
  Runtime int not null,
  FranchiseID int references Franchise(FranchiseID)
);

-- relationship relations

create table Crew (
  CrewID int primary key,
  WorkerID int not null references FilmWorker(WorkerID),
  FilmID int not null references Film(FilmID)
);

create table Cast (
  CastID int primary key,
  WorkerID int not null references FilmWorker(WorkerID),
  FilmID int not null references Film(FilmID)
);

create table WantToWatch (
  UserID int not null references AppUser(UserID),
  FilmID int not null references Film(FilmID)
);

create table Watched (
  WatchedID int primary key,
  UserID int not null references AppUser(UserID),
  FilmID int not null references Film(FilmID)
);

-- multivalued attribute relations

create table CrewJobs (
  Job varchar(20) not null,
  CrewID int not null references Crew(CrewID)
);

create table CharacterNames (
  CharacterName varchar(40) not null,
  CastID int not null references Cast(CastID)
);

create table WatchDates (
  WatchDate date not null,
  WatchedID int not null references Watched(WatchedID)
);
```

As you may have already noticed, certain names of symbols had to be renamed. Notably, The User entity relation has been renamed to AppUser, since ‘user’ is a reserved keyword in SQLPlus. Film’s ReleaseDate attribute has been renamed to ReleaseYear, and now stores an integer rather than the originally planned date attribute type. Changes like these were made to facilitate easier data entry for this proof-of-concept project. Of course, in a real-world application, this would store the full release date.

D. Sample Queries

To illustrate the flexibility offered by this database schema, some sample queries are discussed below. Each output includes the elapsed time, which will be discussed in the following section (E).

1. *Get franchise name, title, and release year of films in The Before Trilogy.*

```
create or replace view q1 as
  select distinct FranchiseName, Title, ReleaseYear as year
  from Film
      left join Franchise
        on Film.FranchiseID = Franchise.FranchiseID
  where FranchiseName = 'The Before Trilogy'
;
```

This query illustrates how a Franchise collects films under a single series. Querying for an entire franchise is as simple as computing a single join. Below is a sample output.

```
FRANCHISENAME         TITLE                 YEAR
-------------------- ---------------- ----------
The Before Trilogy   Before Sunset         2004
The Before Trilogy   Before Midnight       2013
The Before Trilogy   Before Sunrise        1995

Elapsed: 00:00:00.00
```

2. *Get all the crew jobs that Richard Linklater has had in all of their films.*

```
create or replace view q2 as
  select distinct FirstName || ' ' || LastName as name, Job as
credits
  from CrewJobs, Crew, FilmWorker
  where CrewJobs.CrewID = Crew.CrewID
      and Crew.WorkerID = FilmWorker.WorkerID
      and FirstName = 'Richard' and LastName = 'Linklater'
;
```

This query illustrates the flexibility of the database to allow a single film worker to have multiple responsibilities. For example, Richard Linklater has been a director, writer, and producer for a number of his films. Below is a sample output.

```
NAME                  CREDITS
-------------------- ----------
Richard Linklater     Director
Richard Linklater     Producer
Richard Linklater     Writer

Elapsed: 00:00:00.01
```

3. *Get the directors of The Matrix.*

```
create or replace view q3 as
  select distinct FirstName || ' ' || LastName as directors
  from FilmWorker
      full outer join
      (
        Crew
          full outer join Film
      on Crew.FilmID = Film.FilmID
          right join CrewJobs
      on Crew.CrewID = CrewJobs.CrewID
      )
      on FilmWorker.WorkerID = Crew.WorkerID
  where Title = 'The Matrix'
      and Job = 'Director'
;
```

This query demonstrates the ability of this database to have multiple people share a crew job or acting role. In this case, both Lana Wachowski and Lilly Wachowski co-directed *The Matrix*. FDb is able to handle this edge case with robustness. Below is a sample output.

```
DIRECTORS
--------------------
Lana Wachowski
Lilly Wachowski

Elapsed: 00:00:00.01
```

4. *Get the roles that Naomi Watts plays in Mulholland Drive.*

```
create or replace view q4 as
  select distinct CharacterName as role, FirstName || ' ' ||
    LastName as actor
  from CharacterNames
      full outer join
      (
        Cast
          full outer join Film
      on Cast.FilmID = Film.FilmID
          full outer join FilmWorker
      on Cast.WorkerID = FilmWorker.WorkerID
      )
      on CharacterNames.CastID = Cast.CastID
  where Title = 'Mulholland Drive'
      and FirstName = 'Naomi' and LastName = 'Watts'
;
```

Mulholland Drive is a film that contains doppelgangers and characters with multiple names/personalities. For example, Naomi Watts plays both Betty Elms and Diane Selwyn. FDb handles films where actors play multiple roles in an elegant way, using a multivalued attribute relation to enable this uncommon occurrence without overcomplicating normal roles. Below is a sample output.

```
ROLE                 ACTOR
-------------------- --------------------
Betty Elms           Naomi Watts
Diane Selwyn         Naomi Watts

Elapsed: 00:00:00.00
```

E. Performance Analysis

Performance results were included in the output for the queries of the previous section, but unfortunately do not accurately represent the potential performance of this database application due to the lack of data populating the database (less than 100 rows in the entire DB). Evidently, each query takes up to a millisecond to execute. Because my database was designed from the ground up, I was not able to utilize a pre-existing database to populate this one. All data entry was performed by myself, and due to the time limit of the project, I chose to focus on creating an interesting design and useful queries rather than spending hours manually entering film information. Thus, the above timing results are provided for completeness sake, but should be taken with a grain of salt.

However, I will comment on the number of joins many of these necessary queries utilize. While I attempted to optimize the inclusivity of each join, queries like examples 3 and 4 utilize three joints each, which, on a database with millions of entries, may slow down execution time significantly. However, condensing, say, the CrewJobs or CharacterNames multivalued attribute relations into their associated relationships relations would sacrifice their multivalued-ness, destroying the flexibility of my design. Thus, I chose to leave the design as is. While removing dependencies like this is usually possible via normalization, I did not recognize any opportunities to perform these transformations.

IV. CHALLENGES

The major challenge of this project was the tedium of manually entering each and every tuple into the SQL database. While designing the schema itself had its own set of difficulties (as will be discussed next), the data entry took a great amount of time to insert correctly and accurately. Thus, only a small set of films, workers, etc. are included to simply create a proof-of-concept of the working database. I recognize that much of this information could have likely been imported from an existing database, but doing so would come with a significant amount of work in translating the pre existing schema to fit mine. Also, it would have been out of scope of this project's goal to design a database application.

Designing the database schema itself had its own set of challenges, as well. While creating an entity-relationship representation, I often found myself correcting mistakes in order to optimize the translation into a relational schema later down the line. For example, deciding how to structure the Cast, Crew, and FilmWorker relationships and entities was largely dictated by how I would be representing these as relations in SQL.

Another challenge was obtaining realistic performance results. Since my manually-entered database was very small compared to any real-life database with hundreds of thousands if not millions of entries, my performance results were nowhere near a real-life use case. However, simply by analyzing the code in terms of efficiency, it was possible to glean certain qualities from my code.

V. CONCLUSION

This report has summarized the process, methodology, and results from the development of FDb. Overall, the project was a success. The schema was implemented as close to the original design as possible, while still retaining all intended flexibility and robustness to represent the common and edge cases of films, cast members, crew members, etc. Queries were demonstrated to fully utilize the robustness of the schema, accessing data correctly in these edge case scenarios.

The database is complete and robust enough to enable extension into a full web application. As I learn front-end web development in my career, it will be useful to have a well-designed database such as FDb as a backbone from which to create my own web application.

VI. REFERENCES

1. IMDb, https://www.imdb.com/
2. The Movie Database (TMDb), https://www.themoviedb.org/
3. Letterboxd, https://letterboxd.com/
4. Fundamentals of Database Systems, 7th edition, Elmasri and Navathe, 2015