

Portfolio Project one

1. Project Introduction	2
2. Application Sketch	2
2.1 Key features of the Single-Page Application.	2
2.2 Design decisions & the data model.	4
3. The data model	5
3.1 The process	5
3.1.1 Database Model Design	5
3.1.2 Implementation of the database model	6
4. Functions	7
4.1 User Management Functions and Triggers	7
4.2 String Search and Triggers	10
4.3 Movie Rating Functions and Triggers	11
4.4 Structured string search	12
4.5 Finding names	12
4.6 Finding co-players	13
4.7 Name Rating	14
4.9 Similar movies	15
4.10 Frequent person words	15
4.11 Exact-match querying	16
4.12 Best Match Querying	16
4.13 Word-to-words querying	17
5.0 Indexing	18
5. 1 Testing	19
6. Conclusion	20
7. Appendix	21

1. Project Introduction

The primary aim of this sub-project is to architect and construct a resilient and high-performance database system. This will serve as the foundational framework for a single-page web application focused on a movie database. This database will be the central storage for all essential movie-related data. In this Portfolio one section, we will present an initial sketch of the application and outline the key features tied to the movie data model. The sections included in this report will function as documentation but also cover reflection in regards to the development process.

2. Application Sketch

Our application aims to offer a seamless and user-friendly experience similar to IMDb, the well-known movie database. As a Single Page Application, our design focuses on smooth navigation without requiring page reloads, making it convenient for users to browse through a large catalog of movies, TV shows, and related content. The layout has been sketched to prioritize readability and quick access to key features, ensuring that users can easily find what they are looking for. From intuitive search functionalities to detailed movie pages, the design has been crafted to offer a comprehensive and engaging user experience.

It's important to note that these are our team's first thoughts on the design and layout. While we've aimed for readability and quick access to key features, there is still room for adjustments and improvements. We are open to changes that enhance the user experience and the study team is very much aligned on the fact that as we learn and grow there will most likely be improvements to the design.

2.1 Key features of the Single-Page Application.

Our initial design sketches have been focused around several core functionalities aimed at offering an enriched user experience. Below are the key features we envision for our IMDb-inspired Single Page Application:

- **Login & Sign-up**
 - Allow users to create an account or log in to personalize their browsing experience.

- **Advanced Search**
 - Offers various search algorithms such as exact-match, best-match and word-to-word search to help users find exactly what they are looking for.

- **Rate**
 - Enables users to rate movies or TV shows, contributing to a more interactive and user-driven platform.

- **Bookmark**
 - Provides an option for users to bookmark their favorite content for easy access later in their movie search adventures.

- **Explore**
 - A feature aimed at helping users discover new movies, TV shows, and other content based on their preferences and browsing history.

- **User Settings**
 - Enables the user to update their existing data when signing up but also allows the user to personalize their profile.

- **Advanced Listing**
 - Curates lists of popular actors, similar movies, and other relevant content to enhance user discovery.

As a final remark on the key features of the application, it's worth mentioning that this is an initial outline and we are open to making changes to improve the design and functionality as

we progress in the development stage. While we aim and aspire to be similar to IMDB we also find inspiration from other alternative movie sites such as *¹Rotten Tomatoes*.

The design itself can be found in the appendix but there is also provided a link to the design platform and the product itself right here: [Figma board](#)

2.2 Design decisions & the data model.

In addition to outlining the application's key features, the team recognized that these functionalities could have a direct impact on the data model created. For example, the decision to incorporate a login and a sign-up feature necessitated a data model capable of securely storing passwords and other user-specific information. This was because when a said feature user authentication comes in focus. This choice has implications for how we manage data encryption and privacy within our database.

similar considerations were taken into account for all the key features of the application.

Below are some highlighted examples:

- **Rating System:**
 - Implementing a rating feature demands a data model that not only stores the ratings but also associates them with both the user and the specific movie or TV show. This influenced the complexity of the team's queries.
- **Advanced Search**
 - The search options like word-to-word search have implications for how we retrieve data also requiring a more complex query system in our data model.
- **Advanced Listing**
 - The requirement of being able to look after actors in a collection suggests that the team had to have a data-model which could handle and manage many-to-many relationships.

¹ <https://www.rottentomatoes.com/> | Alternate movie site to IMDB

3. The data model

The initial step for the data model was to analyze the provided source data. Although the provided tables could serve as a basic representation, the team discovered design flaws, such as redundancy and lack of normalization. The feel was that we could enhance and optimize the database by creating some changes and so we did.

3.1 The process

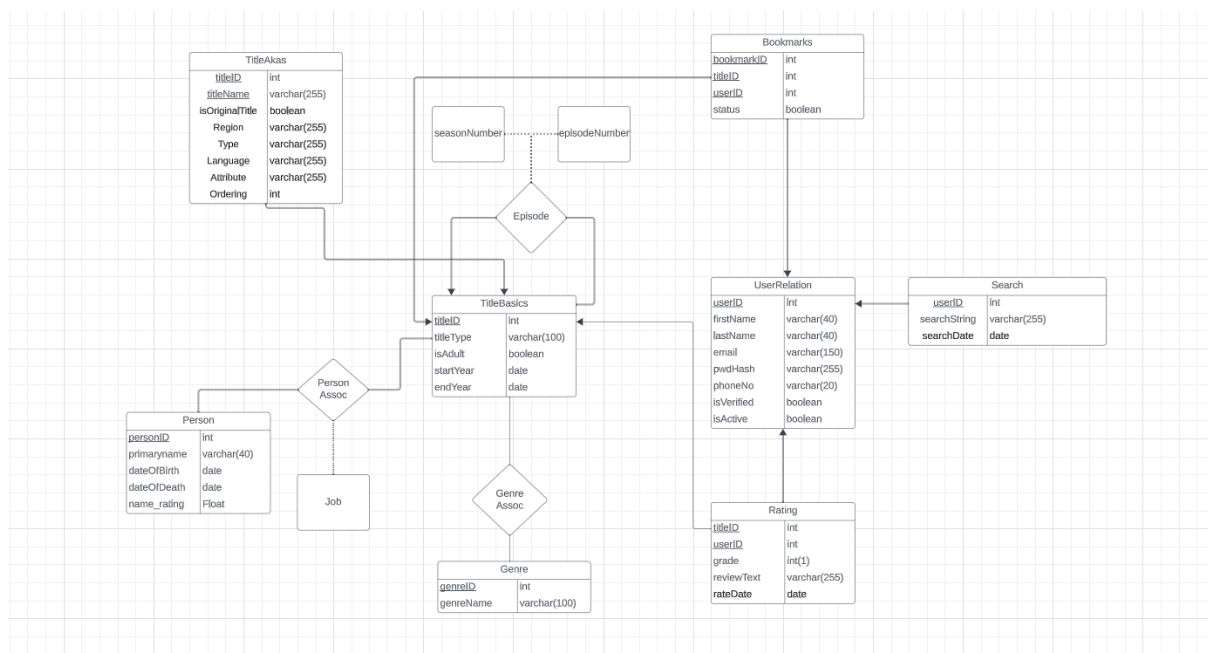
3.1.1 Database Model Design

As part of the process we also had to consider scoping of the project. What did we have to do to meet the requirements of the project but what additional features should we consider in order to meet the desired outcome of the project. As for the data model design we began by identifying the core entities and relationships. At this stage we did not concern ourselves with implementation details in PostgreSQL. During this phase we took both the movie data model and the framework model into consideration.

To eliminate data redundancy and improve data integrity, we applied normalization and did our best to act within the rules of 1NF, 2NF and 3NF. Additionally we discussed appropriate data types for each attribute. This was crucial for storage efficiency and performance optimization.

After our initial discussions and preliminary work -we decided to move towards development of an entity-relationship diagram in order to visualize the model. In figure 3.1 the reader should receive an overview of said ERD but additionally it can also be found in the appendix.

Figure 3.1



Legend:

➔ - Many to One

— - Many to Many

3.1.2 Implementation of the database model

Following the initial phase of the data model where the analytical thoughts were shared and design was constructed, the team moved towards implementation of the database. We started by importing all the source data into a Postgresql database as guided by the CITP note from Moodle.

We then initialized the use of SQL scripts to create new tables that aligned with the conceptual design from the phase before. Doing so we made sure that the data types for each attribute aligned with the design to ensure efficiency and integrity. In the figure below you'll view the commands used to set up the database. It can also be found in the appendix.

Figure 3.2

```

How to run the scripts using postgresql command line:

psql -U postgres -c "create database movie"
psql -U postgres -d movie -f B2_build_movie_db.sql
psql -U postgres -d movie -f C2_build_framework_db.sql
psql -U postgres -d movie -f omdb_data.backup
psql -U postgres -d movie -f imdb.backup
psql -U postgres -d movie -f wi.backup
psql -U postgres -d movie -f Insertions.sql
psql -U postgres -d movie -f functions.sql
    
```

The source data provided was distributed into the new tables and was following that deleted leaving the database in its desired state.

4. Functions

In this subproject, the primary focus is on developing key functionality for a database system that deals with movie data. This functionality will be exposed through an Application Programming Interface (API) composed of functions and procedures implemented in PostgreSQL. The goal is to enable efficient searching, rating, and analysis of movie-related data, while also considering user interactions and maintaining a robust data framework.

The key functionality to be developed includes basic framework features such as user management, bookmarking, adding notes, and retrieving user-specific history. Additionally, there will be a focus on implementing various search capabilities, rating functions, and advanced querying options.

This introduction provides an overview of the tasks outlined in this subproject, emphasizing the importance of creating a comprehensive and user-friendly API for accessing movie data in the database. The subproject also discusses potential improvements in query performance through database indexing and highlights the need for thorough testing to ensure the functionality works as intended.

The following sections will delve into the specific tasks and requirements outlined in the subproject, guiding the development of the API for managing and exploring movie data efficiently.

4.1 User Management Functions and Triggers

Function name:

`create_user(firstName, lastName, email, pwdHash, phoneno, isverified, isactive)`

Description:

This SQL procedure is used to create a new user in the database. It inserts user information into the `userrelation` table, including their first name, last name, email, password hash, phone number (if provided), and verification and activation status.

Input

- `firstName` (varchar(40))First name of the user.
- `lastName` (varchar(40))Last name of the user.
- `email` (varchar(150))User's email address.
- `pwdHash` (varchar(255))Hashed password of the user.
- `phoneno` (varchar(20))User's phone number (optional).
- `isverified` (bool)Indicates if the user's email is verified (default is false).
- `isactive` (bool)Indicates if the user's account is active (default is false).

Procedure name:

delete_user(idUser)

Description:

This SQL procedure deletes a user from the `userrelation` table based on their user ID.

Input:

- `idUser` (int4)The ID of the user to be deleted.

Function name:

update_bookmarks_before_deletion()

Description:

This PL/pgSQL function is triggered before a user is deleted. It deletes all bookmarks associated with the user being deleted. The trigger is activated before a user deletion operation (`DELETE FROM userrelation`).

Procedure name:

bookmark_movie(idUser, idTitle, stat)

Description:

This SQL procedure is used to insert a new bookmark for a user into the `'bookmarks'` table, specifying the user, the movie, and the bookmark status.

Input:

- `'idUser'` (int4) The ID of the user who is bookmarking a movie.
- `'idTitle'` (varchar(255)) The ID of the movie being bookmarked.
- `'stat'` (bool) The status of the bookmark (true or false).

Procedure name:

delete_bookmark_movie(idUser, idTitle)

Description:

This SQL procedure deletes a bookmark for a specific user and movie from the `'bookmarks'` table.

Input:

- `'idUser'` (int4) The ID of the user whose bookmark is being deleted.
- `'idTitle'` (varchar(255)) The ID of the movie for which the bookmark is being deleted.

Function name:

update_rating_before_deletion()

Description:

This PL/pgSQL function is triggered before a user is deleted. It deletes all ratings associated with the user being deleted. The trigger is activated before a user deletion operation (`DELETE FROM userrelation`).

4.2 String Search and Triggers

Procedure name:

string_search_insert(idUser, s)

Description:

This PL/pgSQL procedure inserts a search string along with the user ID and the current date into the `search` table. If the same user has previously searched for the same string, the old entry is deleted before inserting the new one.

Input:

- `idUser` (int4) The ID of the user performing the search.
- `s` (varchar(255)) The search string.

Function name:

string_search(userid, s)

Description:

This SQL function performs a structured search for movie titles and aliases based on a given search string. It also calls `string_search_insert` to insert the search string into the `search` table. It returns movie titles and aliases that match the search string.

Input:

- `userid` (int4) The ID of the user performing the search.
- `s` (varchar(255)) The search string.

Output:

Returns a table with two columns:

- `titleid` (varchar(255))The ID of a movie title matching the search.
- `titleName` (varchar(255))The name of a movie title matching the search.

Function name: update_search_before_deletion()

Description:

This PL/pgSQL function is triggered before a user is deleted. It deletes all search records associated with the user being deleted. The trigger is activated before a user deletion operation (`DELETE FROM userrelation`).

4.3 Movie Rating Functions and Triggers

Function name:

update_movie_rating_fnc()

Description:

This PL/pgSQL function is triggered after a new rating is inserted into the `rating` table. It updates the `movie_rating` field in the `titlebasics` table by calculating the average movie rating based on the ratings given by users.

Procedure name:

rate(tid, pid, grade, reviewText)

Description:

This PL/pgSQL procedure is used to insert a new movie rating into the `rating` table, including the user ID, movie ID, rating grade, and optional review text.

Input:

- `tid` (varchar(255))The ID of the movie being rated.

- ``pid` (int4)`The ID of the user providing the rating.
- ``grade` (int)`The rating grade.
- ``reviewText` (varchar(255))`Optional review text.

4.4 Structured string search

Function name:

`structured_string_search(userid int4, title_name varchar(255), plot_text text, job varchar(255), person_name varchar(255))`

Description:

This SQL function lets you search for movie titles using different criteria like the movie's name, plot, job roles, and people involved. It gives you back a list of movie titles that match what you're looking for. The search doesn't care about upper or lowercase and can find partial matches too.

Input:

`string_search(userid int4, title_name varchar(255), plot_text text, job varchar(255), person_name varchar(255))`

Output:

The function returns a list of unique movie titles that match your search criteria. Each entry in the list includes the movie's ID and name.

4.5 Finding names

Function name:

`structured_search_actors(actor_name, text)`

Description:

This function is designed for structured searching of actors or actresses based on their names. It takes a partial or full actor's name as input and returns a table containing person IDs and primary names of actors or actresses whose names contain the specified text.

Input:

`actor_name (TEXT)`: The partial or full name of the actor or actress you want to search for.

Output:

A table with two columns:

- "personid" (VARCHAR(255)): The unique identifier (person ID) of the actor or actress.
- "primaryname" (VARCHAR(255)): The primary name of the actor or actress.

4.6 Finding co-players

Function name:

find_frequent_coactors(pa_id VARCHAR(255))

Description:

This function is used to find and return the names of frequent co-actors for a given person. It calculates the frequency of collaboration (i.e., working together in the same movies) between the specified person and other actors or actresses in the database. The result is ordered by the frequency of collaboration in descending order.

Input:

pa_id (VARCHAR(255)): The unique ID of the person for whom you want to find frequent co-actors. We choosed to have the id instead of the name of the actor because the primarynames are not unique and without any other information it's impossible to know what it's the desired actor the user wants to do the search on.

Figure 4.1

primaryname	count
John Smith	14
Michael Smith	11
Mike Johnson	9
Eric Johnson	9
Sean Brown	9
Ben Johnson	9
Chris Martin	9
Vijay	9
Steve Jones	8
John Taylor	8

Output:

A table with two columns:

- "co_actor_name" (VARCHAR(255)): The name of the co-actor who frequently worked with the specified person.
- "frequency" (BIGINT): The frequency of collaboration (number of movies worked together) with the specified person.

4.7 Name Rating

Function Name:

update_name_ratings()

Description:

This function is used to update the "name_rating" attribute in the "Person" table. It calculates the weighted average rating titles associated with actors and actresses and assigns this value to their respective "name_rating" field.

Trigger:

A trigger named "update_name_ratings_trigger" is associated with this function. It automatically executes the function after an update operation on the "titlebasics" table.

4.8 Popular actors

Function name:

popular_actor(title VARCHAR(100))

Description:

This function takes the title of a movie or TV show as input and returns a table containing the names of actors or actresses who worked on that title. The result is ordered by the "ordering" column from the "personAssociation" table.

Input:

title (VARCHAR(100)): The title of the movie or TV show for which you want to find popular actors/actresses.

Output:

A table with a single column "actor_name" (VARCHAR(100)) that contains the names of actors or actresses who worked on the specified title.

4.9 Similar movies

Function name:

similar_movie(movie VARCHAR(100))

Description:

This function takes the title of a movie as input and returns a table containing the titles of other movies that share the same genre with the specified movie. It excludes the input movie from the result.

Input:

movie (VARCHAR(100)): The title of the movie for which you want to find similar movies.

Output:

A table with a single column "movie_name" (VARCHAR(100)) that contains the titles of movies similar to the specified movie in terms of genre.

4.10 Frequent person words

Function name:

person_words(person VARCHAR(100), max_length int)

Description:

This function takes the name of a person (actor or actress) and a maximum results length as input. It returns a table containing the most frequently occurring words in titles associated with that person. The result is ordered by word count in descending order and limited to the specified maximum length.

Input:

person (VARCHAR(100)): The name of the person (actor or actress) for whom you want to find frequently occurring words in associated titles.

max_length (INT): The maximum number of results displayed.

Output:

A table with two columns:

- "word" (VARCHAR(100)): The word that appears in titles.
- "counter" (INT): The count of how many times the word appears in titles associated with the specified person.

4.11 Exact-match querying

Function name:

get_exact_match(keywords text[])

Input:

keywords (text[]) - An array of keywords to search for exact matches.

Output:

Returns a table with two columns:

tconst (text) - The unique identifier for a movie.

title (text) - The title of the movie that exactly matches one of the provided keywords.

Description:

This function searches for movies in the database that have exact keyword matches in their titles or aliases. It takes an array of keywords as input and returns a list of movies (tconst and title) where at least one of the provided keywords matches the title or alias of the movie.

4.12 Best Match Querying

Function name:

get_best_match_titles(query_text text)

Input:

query_text (text) - The text query used to search for movie titles.

Output:

Returns a table with two columns:

title (text) - The title of a movie that matches the query text.

rank (bigint) - A ranking value indicating the relevance of the movie title to the query text.

Description:

This function performs a fuzzy search for movie titles based on a given query text. It calculates a ranking based on how many occurrences of words in the query text are found in the movie titles. The results are ordered by rank in descending order, with higher-ranked titles appearing first.

4.13 Word-to-words querying

Function name:

get_word_rank(query_text text)

Input:

query_text (text) - The text query used to search for word rankings.

Output:

Returns a table with two columns:

word_rank (text) - A word from the database that matches the query text.

weight (bigint) - A weight indicating the ranking of the word based on its occurrence in movie titles.

Description:

This function searches for words in movie titles that match the provided query text. It calculates a ranking based on how many times each word from the query text is found in the movie titles. The results are ordered by rank in descending order, with higher-ranked words appearing first.

Please note that these functions appear to be working with tables named wi and titleakas, and they utilize PostgreSQL's PL/pgSQL language for database operations. Additionally,

temporary tables are used within the functions to store intermediate results and are dropped at the end of each function's execution to clean up temporary storage.

5.0 Indexing

In order to further optimize our database and make our functions faster we applied indexing.

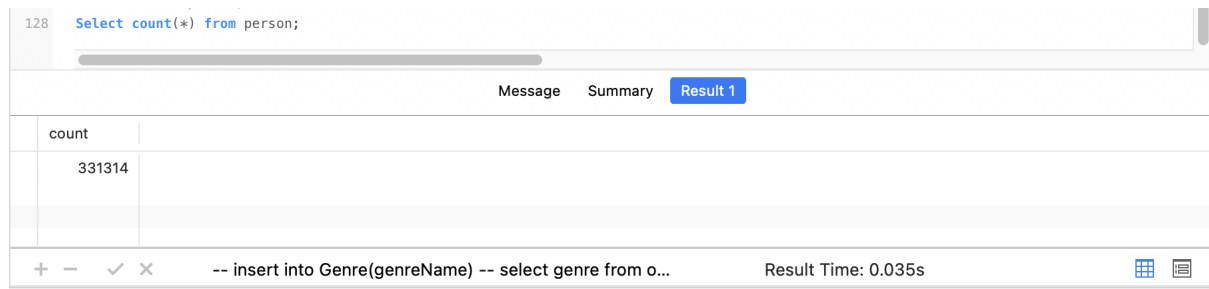
Indexing is a great tool to improve the PostgreSQL query execution.

We created 4 indexes in total:

- `idx_primaryName` from `person`,
- `idx_job` from `personAssociation`,
- `idx_titleName` from `titleakas`
- `idx_word` from `wi`

In order to speed up the person search(actors, actress, directors etc.) we created an index based on the `primaryName`, which contains both first name and last name.

Figure 5.1 | Performance before indexing:



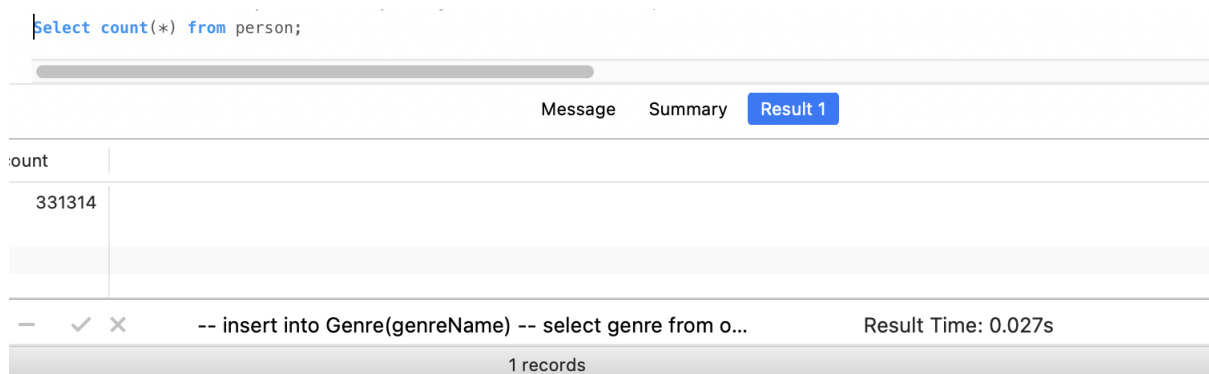
128 `Select count(*) from person;`

Message Summary Result 1

count
331314

+ - ✓ ✕ -- insert into Genre(genreName) -- select genre from o... Result Time: 0.035s

Figure 5.2 | Performance after indexing:



`Select count(*) from person;`

Message Summary Result 1

count
331314

- ✓ ✕ -- insert into Genre(genreName) -- select genre from o... Result Time: 0.027s

1 records

Creating new index

In order to create a new index we used the following code:

```
CREATE INDEX idx_primaryName ON person(primaryname);
```

Retrieving Index

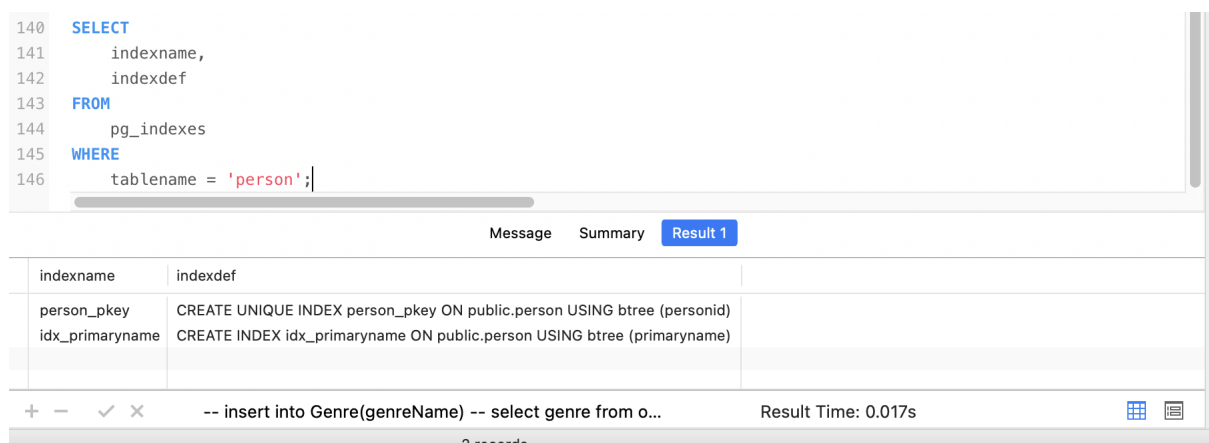
To view our indexes we can search by table or in the whole database to see what indexes we have assigned by using `pg_indexes`

The `pg_indexes`²[2] view allows us to access the index in a table.

The `pg_indexes` view consists of two columns:

- `indexname`: stores the name of the index.
- `indexdef`: stores index definition command in the form of “CREATE INDEX” statement.

Figure 5.3 | Index



```
140 SELECT
141     indexname,
142     indexdef
143 FROM
144     pg_indexes
145 WHERE
146     tablename = 'person';
```

indexname	indexdef
person_pkey	CREATE UNIQUE INDEX person_pkey ON public.person USING btree (personid)
idx_primaryname	CREATE INDEX idx_primaryname ON public.person USING btree (primaryname)

Message Summary Result 1

-- insert into Genre(genreName) -- select genre from o... Result Time: 0.017s

2 records

5.1 Testing

As this was a phase of our project too we defined some clear goals on what was a top priority for us -and that was that the functions we developed were working as planned. Since this was the first part of the project it was important for the study team that they did run without any severe hiccups.

² <https://www.postgresqltutorial.com/postgresql-indexes/postgresql-list-indexes/> | indexing

To make testing easier for us as requested by the assignment specifications we put together a single SQL script (Output.sql). This script included a bunch of SELECT statements and other queries for the purpose of showing data and what it looks like before and after any changes exercised. By doing this we could easily compare and confirm that each function and procedure is doing the rightful job.

6. Conclusion

In summary, this report has focused on the development of a database system, making E-R diagrams, highlighting key observations and recommendations derived from our database creation project.

To begin with, we have created an ER diagram, which contains a framework and a movie model. During the process we had to change it several times due to some mistakes.

Furthermore, we have successfully outlined the fundamental aspects of database design, including data modeling, schema development, and implementation. This process has led to the creation of a robust and structured database system capable of efficiently storing and retrieving data.

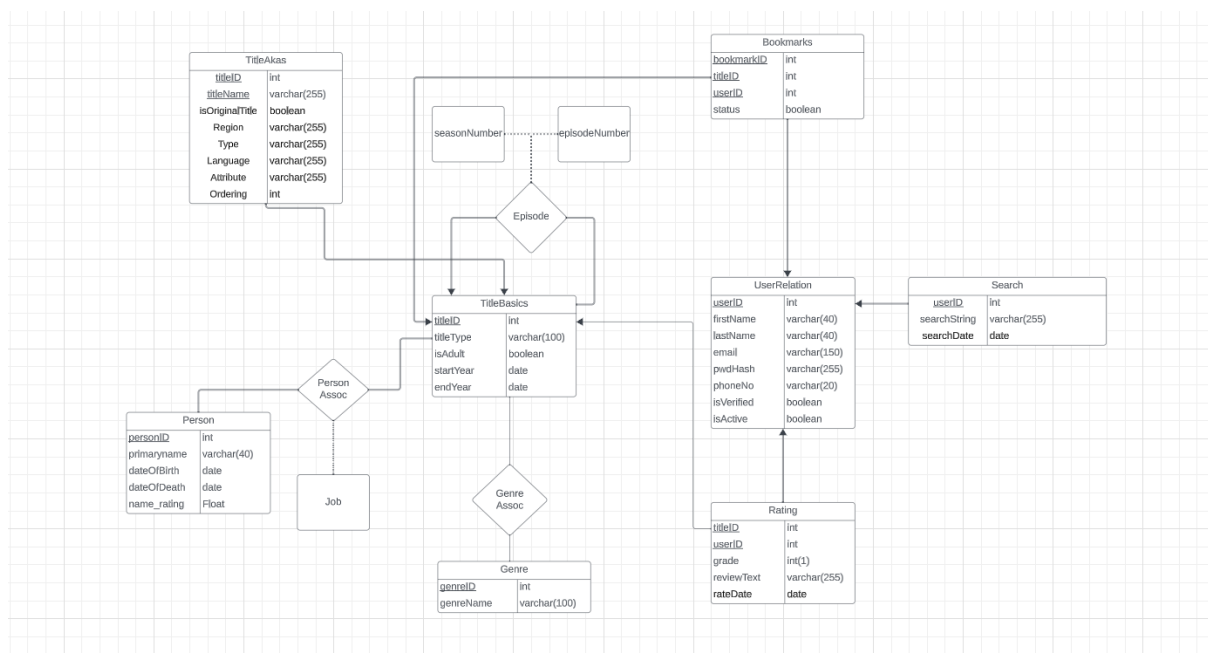
Third, we have filled up our database with data using the source data we have been given, hence we have written several SQL scripts. After the insertions, we dropped the source data, and started to work with our own database. The insertions can be found in the insertions.sql file.

Fourth, we had to write several key functions, deletions and searches. These functions are from D1 - D14. We have effectively written these with lots of communication. These functions can be found in the functions.sql file. We had to write tests for the functions, they can be found in the output.sql file.

Last, we had to work on the performance, therefore we used indexes on our database.

7. Appendix

Figure 3.1 | ER Diagram of the database built.



Legend:

➔ - Many to One

—— Many to Many

Figure 3.2 | Command to run to build the database

```
How to run the scripts using postgresql command line:
```

```
psql -U postgres -c "create database movie"
psql -U postgres -d movie -f B2_build_movie_db.sql
psql -U postgres -d movie -f C2_build_framework_db.sql
psql -U postgres -d movie -f omdb_data.backup
psql -U postgres -d movie -f imdb.backup
psql -U postgres -d movie -f wi.backup
psql -U postgres -d movie -f Insertions.sql
psql -U postgres -d movie -f functions.sql
```

Figure 4.1 | Reflection on a possible bad practice or an issue in the requested function

The screenshot shows a PostgreSQL query editor with the following SQL query:

```
148
149 SELECT primaryname, COUNT(primaryname) from person
150 GROUP BY primaryname
151 ORDER BY COUNT(primaryname) DESC
```

Below the query, there is a table with the results of the query. The table has two columns: 'primaryname' and 'count'.

primaryname	count
John Smith	14
Michael Smith	11
Mike Johnson	9
Eric Johnson	9
Sean Brown	9
Ben Johnson	9
Chris Martin	9
Vijay	9
Steve Jones	8
John Taylor	8

At the bottom of the screenshot, there is a status bar that reads: "-- insert into Genre(genreName) -- select genre from o... Result Time: 0.268s".

Figure 5.1 | Performance before indexing:

The screenshot shows a PostgreSQL query editor with the following SQL query:

```
128 Select count(*) from person;
```

Below the query, there is a table with the results of the query. The table has one column: 'count'.

count
331314

At the bottom of the screenshot, there is a status bar that reads: "-- insert into Genre(genreName) -- select genre from o... Result Time: 0.035s".

Figure 5.2 | Performance after indexing:

Portfolio Project One | Databases

Alex, Bianca, Cristina, Karsten, Lasse | Group 10

```
Select count(*) from person;
```

Message		Summary		Result 1	
count					
331314					

-- insert into Genre(genreName) -- select genre from o... Result Time: 0.027s

1 records

Figure 5.3 | Index

```
140 SELECT
141     indexname,
142     indexdef
143 FROM
144     pg_indexes
145 WHERE
146     tablename = 'person';
```

Message		Summary		Result 1	
indexname	indexdef				
person_pkey	CREATE UNIQUE INDEX person_pkey ON public.person USING btree (personid)				
idx_primaryname	CREATE INDEX idx_primaryname ON public.person USING btree (primaryname)				

-- insert into Genre(genreName) -- select genre from o... Result Time: 0.017s

2 records