Software Documentation

Database Scheme:

Table name	Table columns		Table Keys	
	Name	**Type**	**PK**	**FK**
movie	id	INT	id	None
	title	VARCHAR(256)		
	release_date	DATE		
	runtime	INT		
	description	VARCHAR(1024)		
	rating	FLOAT		
	production_budget	INT		
	marketing_budget	INT		
	revenue	INT		
person	id	INT	id	None
	full_name	VARCHAR(256)		
	birth_date	DATE		
	death_date	DATE		
genre	id	INT	id	None
	title	VARCHAR(128)		
role	id	INT	id	None
	title	VARCHAR(128)		
movie_genre	movie_id	INT	None	• movie_id REFERNCES movie(id) • genre_id REFERENCES genre(id)
	genre_id	INT		
movie_role	movie_id	INT	None	• movie_id REFERNCES movie(id) • person_id REFERENCES person(id) • role_id REFERENCES role(id)
	person_id	INT		
	role_id	INT		

Database Indexes (excluding keys):

- Index on movie.release_date
- Index on movie.rating
- Full-text index on movie.description
- Full-text index on person.full_name

Database Queries:

Query Name	Main/ Secondary	Description	Parameters
query_1	Main	Return a list of persons who had a better than average rating in a specific genre while performing the given role. List ordered by a person's average rating for projects fitting role and genre, in descending order.	• role • genre
query_2	Main	Computes the total revenue for each movie genre since the specified start year (inclusive). Return a list ordered by revenue descending.	• starting release year
query_3	Main	Retrieves the top 100 most active people in a specific role since a given starting release year, ordered by person count of projects.	• role • starting release year
query_4	Main	Retrieves people with a specific string in their full name, along with their date of birth. List ordered by a person's full name ascending.	• search string
query_5	Main	Retrieves movies which have specific strings (up to 3 strings) in their description (AND logic), along with their rating and description. List ordered by a movie's rating descending.	• search string 1 • search string 2 • search string 3
query_6	Secondary	Retrieves all roles.	None
query_7	Secondary	Retrieves all genres.	None

Database optimization:

1. Chosen scheme design:
   We designed our database to accommodate the queries and workflow of our application requires. We have tables for both our main objects of interest - movies and industry workers. They include columns of relevant information in a one-to-one relation so that users can access most relevant information about the object by selecting from a single table.
   We also created tables to accommodate the many-to-many relations – both 'genre' and 'role' are separate tables with unique id columns which we than link using relations 'movie_genre' and 'movie_role' to relevant movies. It also helps us in the application to retrieve the genres and roles efficiently for users when presenting what options are available.
   Alternate design options:
   a. We considered putting data about revenue for each movie in a separate 'movie_revenue' table, using movie.id to link the data. We thought that in practicality this information will be updated more often than static data about the movie. However, because it is a numerical value and not categorical and to avoid

unnecessary joins, we decided to put this information as columns in the 'movie' table.

b. In the original IMDB files we took the information from, the movie's genre was a whole string column which included all relevant genres. Because we noticed the many-to-many relation, we decided to separate them and create a relation 'movie_genre' instead of having a single VARCHAR column in the 'movie' table. That way we don't need to use another Full-text index for our queries. Also we guarantee that the user cannot add/search custom genres that are not recognized as a result of foreign key in the relation.

2. Index optimization:
   Key indexes – the primary and foreign keys optimize the join operations, which we do multiple times for each main complex query because we have separate tables for linking many-to-many relations.
   Full-text indexes - We have 2 full-text indexes, both movie.description and person.full_name to optimize query_4 and query_5. This helps us handle searches on large volumes of text data efficiently just as we saw in class.
   Regular indexes – We filter by year in both query_2 and query_3, selecting movies where their release dates are above a certain year. To optimize those queries, we create an index on movie.release_date and use it for more efficient filtering instead of searching the whole table. Additionally we filter by rating in query_1, selecting by having average above a certain rating we calculate, and order by rating in query_4. To optimize those queries, we create an index on movie.rating for similarly more efficiency.

API Usage:

We used the IMDB public databases in form of .csv files. We also added information from different sources about revenue and movie description to increase options for more diverse queries. We use 'pandas' python module to update our table information from those .csv files.

IMPORTANT: We use CREATE TABLE IF NOT EXISTS in database creation queries so creating an existing table will not cause issues in database and will not produce an error.

Application workflow:

Application 'StudioRanger' is an application designed to streamline the process for film studios when planning and producing new project by providing data on the latest successful movies and working talent in the industry. When opening the application, the user already can see information about the most successful genres in the latest year, to help him decide what genre to focus on in the new project. Next, there are multiple page options, each accommodating specific query to provide helpful info for further planning: Search for active/highly rated talent in the industry to hire; Look up specific person of interest; Search movies by keywords for more specific themes. More information in the user manual…