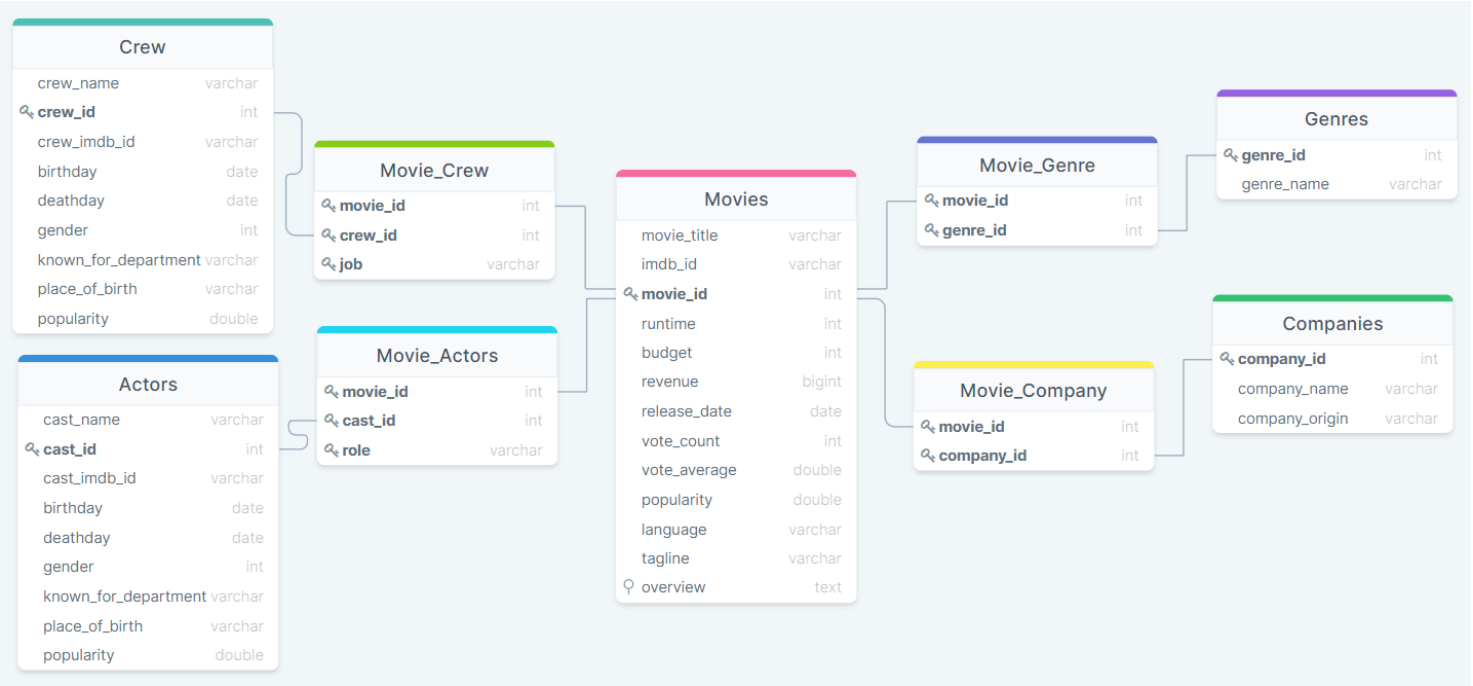
The scheme that we used for our database:



The 5 main tables in our database are:

- **Movies**: this table contains all the movies that are stored in our database (which are 4027 movies), for every movie we store the attributes that are shown in the diagram above. The key of that table is movie_id since its unique for every movie.

- **Genres**: this table contains all the genres of the 4027 stored movies in our first table, there are 19 genres in total and for every genre there is a name and an id. The key for every genre is a number assigned to it called genre_id.

- **Companies**: this table contains all the companies that worked on the movies we have stored, for every company we store its id, name and origin country. The key for every company is a number assigned to it called company_id.

- **Actors**: this table contains all the actors that participated in any of the movies that we have stored, for every actor we store the attributes that are shown in the diagram above. The key for every actor is his id which is called cast_id.

- **Crew**: this table contains all the crew members that participated in any of the movies that we have stored, for every crew member we store the attributes that are shown in the diagram above. The key for every crew member is his id which is called crew_id.

The 4 other tables:

- **Movie_Actors**: since every movie can have multiple actors and every actor can participate in multiple movies, we have a many to many relation between the Movies table and the Actors table, and as we saw in the lectures, the best way for implementing the many to many relations between two tables, is to create a new intermediate table which we called here Movie_Actors and this table will connect the two tables together because it stores the ids of the movies with the ids of the actors that participated in it (we also stored for every actor the role he played in a specific movie), after creating that new table we will have foreign keys from it to both the other tables (Actors and Movies) as it can be seen in the diagram above (the foreign keys in the diagram above are the attributes that the lines connect, for example the line that connects the tables Actors and Movie_Actors is connecting the attributes Actors.cast_id and Movie_Actores.cast_id therefore the cast_id is the foreign key in that case). The key for this table is a tuple of (movie_id, cast_id) since the movie_id alone isn't unique and the cast_id alone isn't unique but the tuple containing both is unique

- **Movie_Crew**: since every movie can have multiple crew members and every crew member can participate in multiple movies, we have a many to many relation between the Movies table and the Crew table therefore we created the intermediate table Movie_Crew, the key of that table is the tuple (movie_id, crew_id) (same as above).

- **Movie_Genre**: since every movie can have multiple genres and every genre can contain multiple movies, we have a many to many relation between the Movies table and the Genre table therefore we created the intermediate table Movie_Genre, the key of that table is the tuple (movie_id, genre_id) (same as above).

- **Movie_Company**: since every movie can be created by multiple companies and every company can create multiple movies, we have a many to many relation between the Movies table and the Company table therefore we created the intermediate table Movie_Company, the key of that table is the tuple(movie_id, company_id) (same as above).

We choose to have these tables since the queries we wrote used a lot of information brought using these tables. The table of Crew and Actors are separated because the API that we used gave us for every movie a list of the Actors and another list for the crew and the difference between them is that the Actor has a role in the movie but the crew member has a job in the movie, and we can see that in the tables Movie_Crew and Movie_Actors since one of them contains the attribute role and the other containts job.

Database optimization using indices:

The overview attribute in the Movies table is a text that describes the movie in a couple of sentences, and in order to search for a word inside it we can optimize that search by using an index on the overview field, and that's what we did for the full-text query, the query gets an input (a word) from the user and it returns the movies that contain that word in their overview, for example if someone wants to search for movies that are about science he can use that query and enter the word "science" and the query will return for him the movies that contain that word in their overview. Instead of searching for the word in the overview using the LIKE word, we use an index on the overview column, and then we can use the MATCH() AGAINST() words which will make the search much faster and that would optimize our query.

Description of the main queries:

(The queries are numbered relatively to the queries in the python file (QUERIES.py))

1. This query sorts the actors according to the number of movies they acted in, in a descending order and outputs the first 1000 actors, this query is simple and very straightforward and there wasn't optimizations to do for it. our DB design supports that query because we can link the Actors table with the Movies table using the intermediate table called Movie_Actors, therefore we will have for every movie the actors that played in it, and then we can just group by the actor name and count the number of movies that he acted in and in the end we sort them in descending order and get the actors sorted starting with actor that participated in the highest number of movies.

2. This query sorts the genres according to the average of (budget/revenue) of all the films of that genre, this query is useful for finding the best genres in terms of making money because if the budget/revenue is a small number it means that the revenue of the movie is much more than the budget and that is a good thing! Therefore we sort the movies according to the budget/revenue from lowest to highest in order to find the most money making genre. For optimizing that query, the first thing we do is take the Movie table and add to it a new column for the budget/revenue, and drop from this table everything else, and then we calculate the Cartesian product between the resulting table with the other tables, that would be much better than first calculating the Cartesian product and then filtering because there are a lot of data stored in the movies table that are irrelevant. The DB supports this query because we can link the Movies table with the Genres table using the intermediate table named Movie_Genre, therefore we will have for every genre its movies and there budget/revenue, then we can group by on the genre name and then we calculate the average value of budget/revenue for every genre and sort the genres according to that.

3. This query finds for every genre, the name of the company that created the highest number of movies in that genre, it is very useful for a director that wants to create a movie with a specific genre and he wants to find a company that has experience in that genre. Our DB design supports this query because we need to link the Genre table to Company table, and we can do that using the Movie_Genre, Movies, Movie_Company tables. The way we implemented this query is, first we created a view named temp which holds for every pair of genre and company the number of movies that this company created that were from that genre (using group by on the genre and company), then we use that view in our query in order to take the maximum over the number of movies for each genre while maintaining the whole row, we do that by taking all the rows that have the maximum as their count (you can see the implementation in the queries file)

4. This is the full-text query that we mentioned above, we take input from the user which is a word, and we return all the movies that have that word in their overview. We optimized this query using an index on the overview column, we explained that above in the database optimization section. For that query we use the table of Movies in our DB design since it contains all the relevant information.

5. This query takes an input from the user which is an actor name, and it outputs the best genres that he played in, according to the average ratings of the movies in that genre that he played in, this is useful because if someone wants to create a movie and wants the actor X to play in it, then he can use this query with input as X and then he will get the best genres for that actor to play in sorted from the best to worst(we return only the genres that he have average rating > 6 in them). For optimization, since the Actors table is huge, we want to filter it first and then take a Cartesian product, because otherwise the product will be enormous. Our DB design supports this query because we can link the Actors table to the Genres table using the Movie_Actors, Movies, Movie_Genre tables. After we take the Cartesian product of these tables we filter the relevant data, and then we group by on the genre name, and calculate the average on the vote_average attribute of the Movies table, then we use the HAVING word to filter the genres that have avg < 6 and then we sort the remaining rows according to their avg in descending order.

6. This query takes an input from the user which is a genre name, and it outputs the top 10 movies in that genre according to their rating(using only movies that have atleast 7000 votes in order to have a rating because we also want the movie to be popular) this feature is very useful since everyone can check what the best movies are in the genres that they love. For optimization, instead of doing the Cartesian product first and then taking only the rows with #votes > 7000, we do that before calculating the Cartesian product in order to filter a lot of rows and then doing the product, also we filter the genre table first and keep only the genre that the user wanted and then do the Cartesian product. Our DB design supports that query because we can link the Genres table to the Movies table using the Movie_Genre table, after calculating the product we filter the irrelevant data, and then order the movies in descending order according to their rating.

7. This query takes an input from the user which is a genre name, and it outputs the top 100 popular actors according to the number of high rated movies in that genre that they acted in(in other words, it sorts the popular actors according to the number of good movies that they acted in), this query is very useful for movie creators that want to create a movie of genre X, and they want to find the perfect actor for their movie. For optimization, first we took only the popular actors from the Actors table and then we did the Cartesian product, we did that also for the genres(we kept only the genre that the user entered, and we kept only movies with high rating (> 6.5)). Our DB design supports this query because we can link the Actors table with the Genre table using the Movie_Actors, Movies, Movie_Genre tables, then we filter the irrelevant

data, then we group by the actor name and count for each actor the number of movies, then we sort according to that count in descending order.