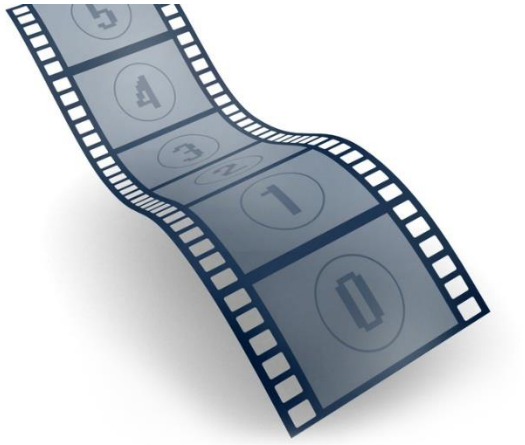
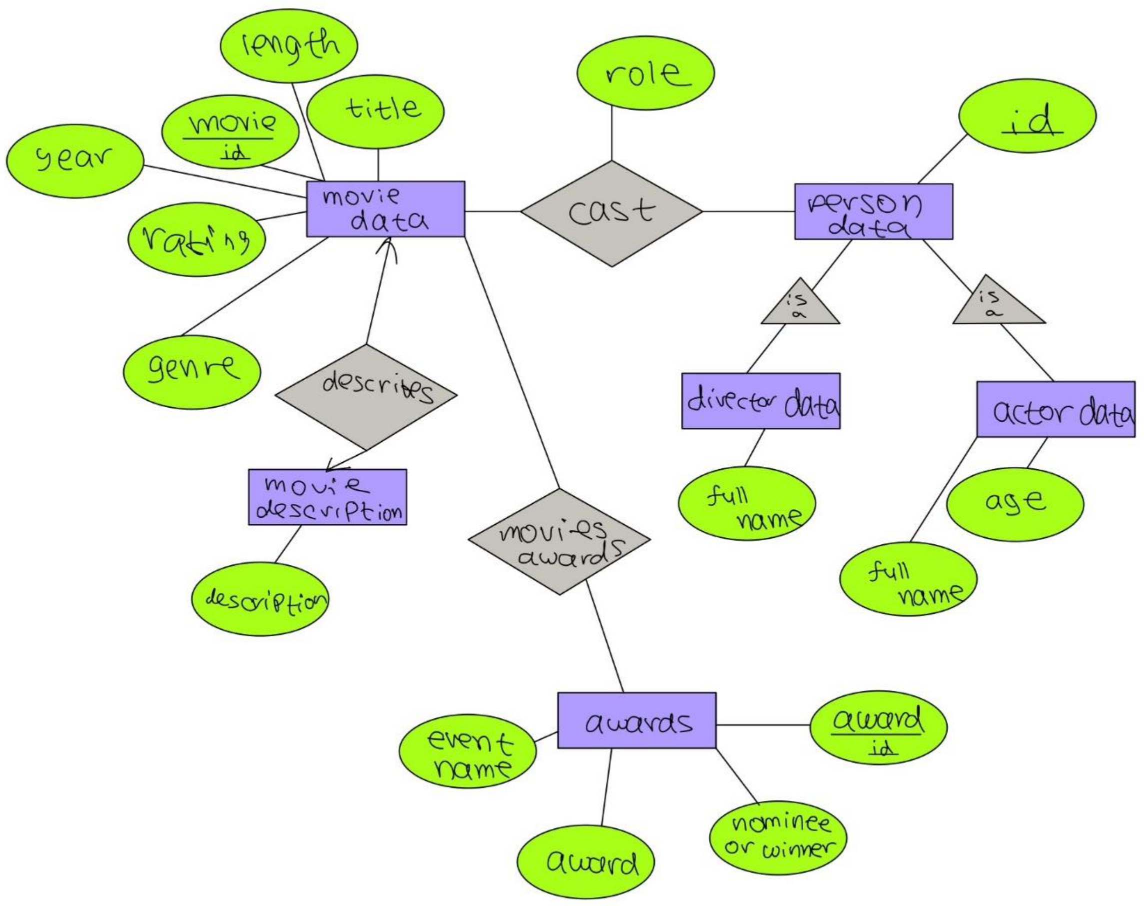
SOFTWARE DOCUMENTATION



DB scheme structure

E/R Diagram:



Scheme description:

Our Data Base consists of 8 different tables. While designing the DB we made a few choices for each table:

- **Movie_data:**
  - First, we chose to add to our data base movies whose rating is higher than 6. Second, we chose to insert only movies from the years 2013-2021, since older movies are less relevant for todays "best sellers'" statistics.
  - In this table, we have the **movie_id** attribute, which is our primary key.
  - Moreover, we added to each movie it's **title**, so the query results will be more meaningful to our producer.
  - When building our data base, we chose to add the **main genre** of the movie. Also, we excluded irrelevant genres that where in the original API such as "talk shows", "news" etc.
  - We chose to add the **length** of the movie, so our producer can see what the average time of movies of different genres is. You surely don't want to make a comedy movie that is too long for the viewers to watch!
  - Each movie contains information about its release **year**, so we could take in account its relevance to the analyzing process.
  - We chose to add the **rating** of the movie. That helps us to evaluate the movie success. We count on the wisdom of crowds!
  - We saved all those attributes in one table, as most of our queries use couple of them. That way, we benefit from faster retrieval of the data.

- **Movie_description:**
  - Here, we also have the **movie_id** attribute, which is our primary key.
  - We added the **description** attribute in order to let our user search key words in the description of the movie. Do you want

your movie to be the next Harry Potter? Look for movies that has "magic", "wizards" etc.

- We chose to separate this attribute from the “movie_data” table because the description attribute of each movie takes a lot of space and memory, and therefore we don’t want to import it every time we use the movie_data table.
- That being said, we do want a reliable connection between the two tables. Therefore, we determined the description(movie_id) to be a foreign key of movie_data(movie_id)

🎬 **Person_id:**

- Contains an **person_id** attribute, which is our primary key.
- Will be a “Super Class” for two different “kinds” of persons – actors and directors.

🎬 **Actor_data:**

- Contains the **actor_id** (this is the primary key) and his **full name**.
- When casting an actor to a specific role, one often wants to take in account some personal information, such as the **age** of the actors.
- This table is a “sub class” of the Person_data table. Every actor **is a** person. Thus, we determined that the actor_id is a foreign key of person_data(person_id).

🎬 **Director_data:**

- As we only care about the director’s professionality, we saved in this table only the **director_id** (this is the primary key) and his **full name** for easier acknowledgment.
- Like the actor_data table, this table also is a “sub class” of the Person_data table, and every director **is a** person. Thus, here also director_id determined as the foreign key of person_data(person_id).

- We chose to separate the actors from the directors, since in most of the queries we look for only one of the professions. That way, we don’t need to search in a large table that contains all the people.

🎬 **"Cast":**

- Here we have the **person_id** and the **movie_id**. Together they are the primary key, which determines the third attribute, **role**, unequivocally.
- We chose to take max of 8 people from the cast list. We also wanted the save the roles, so our producer can choose wisely who to hire for in his new movie. We took information about both actors and directors.
- Since this table contains information about both movies and personals, it has two foreign keys:
  1. actor_id is a foreign key of person_data(person_id).
  2. cast(movie_id) is a foreign key of movie_data(movie_id).

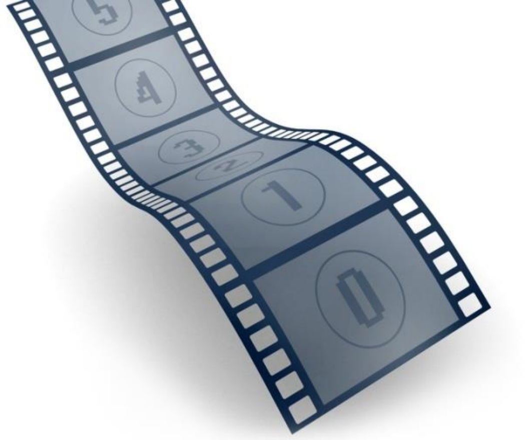
🎬 **Awards**:

- We want our producer’s movie to be the best, so he could be remembered by it. We therefore analyze the successfulness of a movie based on the awards it won and/or was nominated to.
- In this table, we save information about the awards themselves. For every award we have an **award_id**, which determines the **event name**, the **award** given, and the **win or nominee** status. The award id is the primary key in this table.
- When getting the data from the API, we saved only “positive” awards, meaning awards like “worst movie” were not saved in the table. We did that as those are less relevant for our end-user and queries, who look only at the best of each field.

🎬 **Movie_Awards:**

- The purpose of this table is to connect between the movies and the awards they won or were nominated to.
- We have two attributes, **award id** and **movie_id**, both are primely keys in the table.
- There are also foreign keys from other tables:
  1. award_id is a foreign key of awards(award_id).
  2. movie_id is a foreign key of movie_data(movie_id).
- The award_id wasn't originally in the API and was created by us for the purpose of splitting the award information and the winning movies into two different tables.
- We chose to divide the movie awards information into two table, to save memory and space. With our structure, we save for each movie only an award id, instead of saving all the award details. In addition, the awards table is smaller than the movie_award one, since two different movies can win the same award in different years. Moreover, several different movies are nominated (and not winners) for the same award.

Before arriving to this complete table setup, we considered more designs. At first, we had one table which contained the movie ids and all the awards the won. Later we understood that dividing this table to two smaller ones saves up memory and space. In addition, we thought about dividing the movie_data table, which has 6 columns, to several smaller tables. However, we saw that it's not that effective for our choice of queries, since most of the time we would have joined the sub tables anyway, and that way have slower performances.



DB optimization:

We optimized our Data Base in two main ways:

1. By the way we chose to divide the data into tables.
2. By selecting wisely primary and foreign keys.
3. By creating indexes on chosen attributes.

We discussed the benefit of our chosen schema, including our primary and foreign keys, in the "DB scheme structure" chapter.

In general, when trying to decide which indexes we want to add to the DB, we considered several facts:

1. Indexes take more storage and slow down updates. Therefore, we use them only if there is a concrete query that can benefit from them.
2. Indexes that implemented using Hash are better for "a specific value" searches. Clustered indexes speed up the search when looking after ordered data (range).
3. The primary key in each table already has an associated index for faster performances.

We wanted to created indexes on the following attributes:

- **Movie_data(rating):**
  We created a clustered index on this column since in many of our queries we evaluate the successfulness of a movie based on its rating. When selecting movies based on this criterion, we often interested in a range of rating. Thus, having a clustered index helps us to speed up the search for sorted records.

- **Movie_description(description):**

We created a reverse index as we are conducting a "full text query" on this column.

- 🎬 **Actor_data(age):**
  We created a clustered index on this column since we are interested in an age range. Having a clustered index helps us to speed up the search for sorted records, and therefore helps to optimize the query.

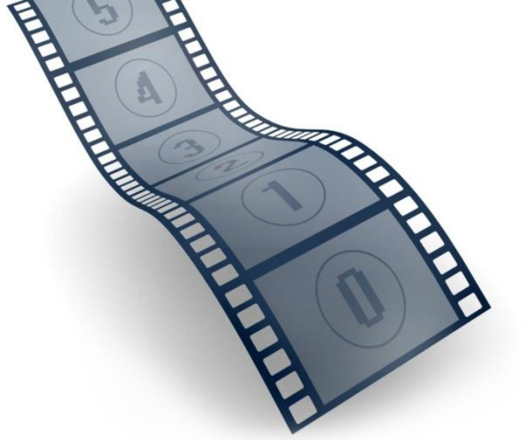
- 🎬 **Cast(role):**
  We created a hash index on this column since in the only query we're using this table, where looking only on a **specific** role. Therefore, putting an index on it and finding it more quickly and efficiently, optimizes our DB.

- 🎬 **Awards(event_name):**
  - We created a hash index on this column since it helps us to optimized query number 6, in which we're looking at awards from a certain event (the query will be described in detail in the next chapter).
  - We also use quite frequently the award_id column, however it is a primary key, and therefore has an index optimization by default.

- 🎬 **Awards(nomineeORwinner):**
  - We created a hash index on this column since it helps us to optimized queries 2, 5 and 6, which are searching for winning movies only.

In practice, we were not able to implement clustered index and hash index on our data base using MySQL.
Specifically for clustered index, we know that there can be only one clustered index in each table. We saw that the primary key is clustered by default, and therefore in some engines there is a possibility to first drop

this index and then add a clustered index on another attribute. However, this technique didn’t work in the MySQL environment.
Even though we were not able to implement it, we discussed here the optimal index setup in our opinion, and that was taken in account during the analysis of the indexes benefits, to show and discuss their advantages.



Queries

Query #1: Find the 3 most popular film genre among films who were release in the last 5 years:

```
SELECT  DbMysql35.movie_data.genre, COUNT(*) AS number_of_movies
FROM DbMysql35.movie_data
WHERE DbMysql35.movie_data.year >=2017
AND DbMysql35.movie_data.rating>=7
GROUP BY DbMysql35.movie_data.genre
ORDER BY COUNT(*) DESC
LIMIT 3
```

With this query, we searched for the most popular film category in the last five years. We took in account only films that we evaluated as "successful" – films who's ranking is above 7. Then, we returned the most common film category among the movies.

We optimized the query by adding a sorted clustered index on the rating attribute, and that way optimized our selection prosses.

We can see that our DB supports this query, as all the general movie data is in the same table.

Query #2 : Find the most successful movies from the specific genre:

```
SELECT DbMysql35.movie_data.title
FROM DbMysql35.movie_data as movie_data
WHERE movie_data.rating >=7.0
AND movie_data.genre = %s
AND movie_data.movie_id IN(
        SELECT DbMysql35.movies_awards.movie_id
        FROM DbMysql35.movies_awards
        INNER JOIN DbMysql35.awards ON
        DbMysql35.movies_awards.award_id = DbMysql35.awards.award_id
        WHERE DbMysql35.awards.nomineeORwinner = "Winner"
        GROUP BY DbMysql35.movies_awards.movie_id
        HAVING COUNT(*)>=3)
ORDER BY  movie_data.title
LIMIT 0,10
```

We want to present to our producers what were the resent best seller, so they'll be able to investigate them forward. Therefore, we return to them the best movies for his category of interest.

To determine which movies are victorious, we want to see their popularity both within the critics and the viewers. Thus, will include in our analysis both the award and the rating a movie got. We return only the most awarded movies that rating is above 7. We won't return movies who got a total of less them 3 awards at all.

This query also benefits from adding an index on the rating attribute. In addition, we can notice that the query is well supported by the DB schema we chose, since all the are connected via foreign keys, that assures us data consistency. In addition, the hash index that was added to the nomineeORwinner attribute helps us to speed up the search as well.

Query #3 : Find the average film length for each film genre:

```
SELECT DbMysql35.movie_data.genre,
AVG(DbMysql35.movie_data.length)                                    as
Average_movie_length_in_minutes
FROM DbMysql35.movie_data AS movie_data
WHERE movie_data.rating >=7
AND movie_data.movie_id IN(
        SELECT DbMysql35.movies_awards.movie_id as movie_id
        FROM DbMysql35.movies_awards as movies_awards
        INNER JOIN DbMysql35.awards ON movies_awards.award_id =
        DbMysql35.awards.award_id
        GROUP BY movie_id
        HAVING COUNT(*)>=3
)
GROUP BY movie_data.genre
ORDER BY  movie_data.genre
```

We want to understand what causes a movie to be a best seller more explicitly. One characteristic we want to investigate is the average length successful movies. We don't want the audience to "not have enough", but we don't want to bore them as well.

In addition, we know that different genres usually have different length – for example, an action film is longer that a comedy.

In the query, we first looked at films who's rating is high – above 7. In addition, we looked only at films that won or were nominated for at least three award – we want to take example from prestige movies! Then, we calculated the average length of the movies for each category.

We optimized this query by adding an index on the rating attribute, and that way optimized our selection prosses.

We can see that our DB supports this query since all our movies have a length attribute as their general data, and those can easily be connected using a unique id key to find out the number of awards the won.

Query #4 : Find the film directors whose film average rating is the highest and released more than 5 movies:

```
SELECT DbMysql35.director_data.full_name
FROM DbMysql35.director_data
INNER JOIN DbMysql35.cast ON DbMysql35.director_data.director_id =
DbMysql35.cast.person_id
INNER JOIN DbMysql35.movie_data ON DbMysql35.cast.movie_id =
DbMysql35.movie_data.movie_id
WHERE DbMysql35.cast.role = "Director"
AND DbMysql35.cast.person_id IN (
        SELECT DbMysql35.cast.person_id AS id
        FROM DbMysql35.cast
        GROUP BY id
        HAVING COUNT(*)>=5)
GROUP BY DbMysql35.director_data.director_id
ORDER BY avg(movie_data.rating)
LIMIT 5
```

In the following query, we wanted to search for a director for our film. To find the best guy for the job, we searched for the one who's done the most successful films. He's going to make our customer's film successful too!
To make sure our director doesn't lack in experience, we require that he's directed at least five different movies.
To optimize the query, we added a hash index on the "role" attribute, so we can easily and efficiently search explicitly for director. Notice that we can benefit from this index if we want to look for different roles as well.
We can see that our DB supports this query, since we can easily connect between the film ranking, the persons role and his name using unique ids, that guarantees correctness. Moreover, those ids are foreign keys from different tables, which also confirm our data integrity.

Query #5: Find the best actors for a given age range:

```
SELECT DbMysql35.actor_data.full_name
FROM DbMysql35.actor_data AS actor_data
WHERE actor_data.age BETWEEN %s AND %s
AND actor_data.actor_id IN (
        SELECT DbMysql35.cast.person_id
        FROM DbMysql35.cast
        INNER JOIN DbMysql35.movies_awards ON
        DbMysql35.cast.movie_id = DbMysql35.movies_awards.movie_id
        INNER JOIN DbMysql35.awards ON
        DbMysql35.movies_awards.award_id = DbMysql35.awards.award_id
        WHERE DbMysql35.awards.nomineeORwinner = "Winner"
        GROUP BY DbMysql35.cast.person_id
        HAVING COUNT(*)>=3
        ORDER BY COUNT(*) DESC)
LIMIT 0,20
```

Now, after providing our producer his director, we want to help him find his stars.

When casting an actor to a specific role, one usually wants to consider his age. You wouldn't hire an elderly actor to play spiderman, no matter how much experience he has. In this query our client can insert the age range he's looking for, and we will return him the most awarded actors in the range.

Of course, as we want only the best of the best – we will consider only actors which already performed in successful movies, as they appeared on at least 3 awarded movies.

We optimized this query by adding a hash index to the nomineeORwinner attribute, that enables us to do an efficient search after the "winners".

The DB schema supports this query since different tables are connected logically to each other via foreign keys and guarantee us data integrity.

Query #6: Find 5 Most Awarded Movies For A Given Award

```
SELECT DbMysql35.movie_data.title AS movie_name
FROM DbMysql35.movie_data as movie_data
WHERE movie_data.movie_id IN (
        SELECT DbMysql35.movies_awards.movie_id
        FROM DbMysql35.movies_awards as movies_awards
        INNER JOIN DbMysql35.awards ON movies_awards.award_id =
        DbMysql35.awards.award_id
        WHERE DbMysql35.awards.event_name = %s
        AND DbMysql35.awards.nomineeORwinner ="Winner"
        GROUP BY movies_awards.movie_id
        ORDER BY COUNT(*) DESC)
LIMIT 5
```

With this query we give support in the case which the user wants to aim for a specific award. For example, some producers dream is to create an Oscar-winning film, others may consider a kid's movie profitable only if it won the kids choice award, and so on.

In this query we're presenting to our producer the top 5 movies, which won the greatest number of prizes in an event of his choice.

We optimized this query by adding a hash index to the event name attribute and nomineeORwinner attribute, thus guaranteeing an efficient search. In addition, we can see that our DB supports this query in a well-defined way, as we created a table that its exclusive goal is to match between awards and the movies who won them.

Query #7 – A “Full Text Query”: Find Movies Based On Description Search

```
SELECT *
FROM DbMysql35.movie_description
WHERE MATCH(description) AGAINST(%s)
```

This query allows the user to search for words in a movie description. It can help our producer to take inspiration from movies in his field of interest.
We optimized this query by adding a reverse index to this column. In addition, our DB supports this query as we separated the description attribute, which contains a lot of data and therefore takes a lot of storage place. We put it in a separate table which will be in use only when explicitly needed. Moreover, the movie_id is a foreign key and connects this table to the movie_data table. That way, when taking the movie_title from the latter, we ensure data integrity.