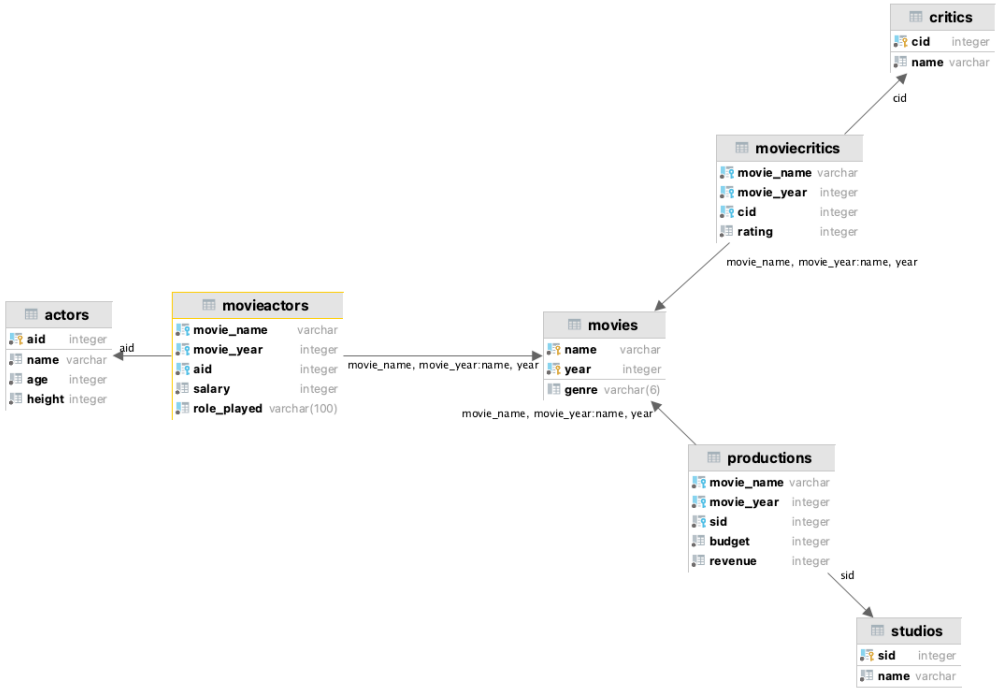
DATABASES HW2

ELIE NEDJAR: 336140116
BENJAMIN SULTAN: 931190987

The database design is described below



We have 4 basic tables which are describing Movies, Actors, Studios, and critics.
For each one the fields are exactly the same as the class described in the file.

Actor:

- aId (Integer): the id of the actor, it's a primary key because first, it must be unique and secondly creating an index on this field will accelerate the queries and because it will be used as a foreign keys in some tables and finally it must not be null.
- Name (Varchar): A not null string describing the name
- Height (Integer): A not null integer describing the height of the actor, with the constraint height > 0
- Age (Integer): A not null integer describing the age of the actor, with the constraint age > 0

Movies:

- Name (Varchar): The name of the movie (more on this one next)
- Year (Integer): The year of the movie's release. A movie is identified with the fields name and year, so we decided to define the pair as the primary for the same reasons as aId in actors
- Genre (Varchar(6)): The string that describe the genre of the movie. We used a constraint to ensure that it's one of the following (Drama, Action, Horror, Comedy)

Studio:

- sId (Integer): The Id of the studio. A studio is identified with his Id, so we define it as a primary key for the same reasons as above.
- Name(Varchar): the name of the studio, define as Not Null

Critics:

- cId (Integer): The Id of the critic. A critic is identified with his Id, so we define it as a primary key for the same reasons as above.
- Name(Varchar): the name of the critic, defined as Not Null

Moreover, as we can see on the diagram, the first described tables are related by some tables that allows us to respond to the needs of the basic API operation such as ActorPlayedInMovie, CriticRatedMovie, StudioProducedMovie (and the ones with "didn't")

MovieActors:

- Movie_name (Varchar): Name of the movie where the actor played
- Movie_year (Integer): Release year of the movie where the actor played, it's a foreign key together with Movie_name, that are referencing name and year of the table Movie
- aId (Integer): Id of the actor that is playing in the corresponding movie. It's a foreign key that is referencing to the aId field of the Actors table
- Salary (Integer): The salary of the actor for this movie, in order to ensure that the salary is not negative we put the constraint Salary > 0 for this field.
- Role_played (Varchar): One of the role that the actor play in the movie

Note: If an actor plays two different roles in the same movie, there will be two different rows for this actors and movie with two, each one having a different role. The unique constraint is on the fields (Movie_name, Movie_year, aId, Role_played)

MovieCritics:

- Movie_name (Varchar): Name of the movie that the critic rated
- Movie_year (Integer): Release year of the movie that the critic rated, it's a foreign key together with Movie_name, that are referencing name and year of the table Movie
- cId (Integer): Id of the critic that rated the corresponding movie. It's a foreign key that is referencing to the cId field of the Critics table
- Rating(Integer): The rate that the critic rated the movie. It's defined to be between 0 to 5

Note: A critic can rate a movie only one time, so the unique constraint is on the fields (Movie_name, Movie_year, cId)

Productions:

- Movie_name (Varchar): Name of the movie that the studio produced
- Movie_year (Integer): Release year of the movie that the studio produced, it's a foreign key together with Movie_name, that are referencing name and year of the table Movie
- sId (Integer): Id of the studio that produced the corresponding movie. It's a foreign key that is referencing to the sId field of the Studios table
- Budget(Integer): The budget of the studio for the movie, the budget isn't negative, so we put the constraint Budget >= 0
- Revenue(Integer): The revenue of the studio for the movie, the revenue isn't negative, so we put the constraint Budget >= 0

Note: A movie can be produced by only one studio, so the unique constraint is on the fields (Movie_name, Movie_year)

For each of the tables production, movieCritics, movieActors, we add the ON DELETE CASCADE constraint so that if a movie, actor, studio or critic is deleted from the one of the basic tables it will be also deleted from the table making a reference to it.

To accelerate queries of the APIs, we have created some views:

- ActorMovieNoRoles : It is the same as MovieActors except that each actor appear only once for each movie he plays in, without the roles.

```
'CREATE VIEW ActorsMoviesNoRole AS '
' SELECT movie_name, movie_year,aid,salary '
' FROM movieactors '
' GROUP BY movie_name, movie_year,aid,salary;'
```

- AverageRating : Contains each movies that has been rated at least once and the average grade on of it.

```
'CREATE VIEW AverageRating AS '
'  SELECT genre, movie_name,movie_year, avg(rating) as avg_rating '
'  FROM moviecritics mc '
'  JOIN movies m on mc.movie_name = m.name and mc.movie_year = m.year '
'  group by movie_name, movie_year, genre;',
```

- AvgRatingActors: Contains for each actors that played in at least one movie the average rating of the movie. If the movie wasn't rated the given average is 0

```
'CREATE VIEW avgRatingActors AS '
'    SELECT ma.aid, ma.movie_name, ma.movie_year, avg_rating, ag.genre'
'    FROM averagerating ag '
'    JOIN ActorsMoviesNoRole ma ON ag.movie_year = ma.movie_year AND ag.movie_name = ma.movie_name '
'UNION '
    'SELECT ma.aid, ma.movie_name, ma.movie_year, 0 as avg_rating, m.genre '
    'FROM moviecritics ag '
    'RIGHT OUTER JOIN ActorsMoviesNoRole ma ON ag.movie_year = ma.movie_year AND ag.movie_name = ma.movie_name '
    'JOIN movies m on ma.movie_year = m.year and ma.movie_name = m.name '
    'WHERE ag.rating IS NULL;',
```

- actorStudio: The view contains for each actor that plays in a movie that has been produced by any studio, the actor and the studio.

```
'actorStudio': 'CREATE VIEW actorStudio AS'
               '    SELECT aid, sid '
               '    FROM ActorsMoviesNoRole am '
               '    JOIN productions p on p.movie_name = am.movie_name and p.movie_year = am.movie_year;',
```

- MovieProdActors: The view contains for each movie the budget of the production and the sum of the salaries of the actors, If one of the above fields doesn't exists there is a Null value in the corresponding columns

```
'CREATE VIEW movieprodactors AS'
' select m.name, m.year, budget, sum(salary) as sum_salaries '
' from movies m '
' LEFT OUTER join productions p on m.name = p.movie_name and m.year = p.movie_year '
' LEFT OUTER join actorsmoviesnorole a on p.movie_name = a.movie_name and p.movie_year = a.movie_year'
' group by m.name, m.year, budget;'
```

CRUD API :

For this we used basic SQL queries such as:

INSERT INTO Table(col1,…,colN) VALUES (val1,…,valN)
DELETE FROM Table WHERE col1 = val1 AND … AND colM = valM

The appropriate return value is returned thanks to the Exception mechanism in python and the constraints we have defined in the tables.

BASIC API:

- AverageRating:

```
query = SQL("SELECT avg_rating "
            "FROM averageRating "
            "WHERE movie_name = {m_n} and movie_year = {m_y};").format(m_n=movieName, m_y=movieYear)
```

From the view averageRating we select the rating of the corresponding movie

- AverageActorRating

```
SQL("SELECT avg(avg_rating) FROM avgRatingActors WHERE aid = {aid};").format(aid=aid)
```

From the view AvgRatingActors we select the average of the column avg_rating of the corresponding actor

- BestPerformance:

```
"SELECT movie_name,movie_year, genre "
"   from avgRatingActors "
"   where aid = {aid} and avg_rating >= ALL"
"       (SELECT avg_rating FROM avgRatingActors where aid = {aid}) "
"   ORDER BY movie_year, movie_name DESC"
"   limit 1;").format(aid=aid)                                    `
```

In the view avgRatingActors we have all the averages of the actors, so if we use the ALL keyword in the where constraint on average of the same table, we get the maximum average among all the actors. For the tie breaker we use the ORDER BY keyword

- StageCrewBudget

```
"SELECT budget - sum_salaries as stage_budget "
"FROM movieprodactors "
"WHERE budget IS NOT NULL and sum_salaries IS NOT NULL "
"AND name = {m_n}  AND year = {m_y} "
"UNION "
"SELECT 0 as stage_budget "
"FROM movieprodactors "
"WHERE budget IS NULL "
"AND name = {m_n}  AND year = {m_y} "
"UNION "
"SELECT budget as stage_budget "
"FROM movieprodactors "
"WHERE sum_salaries IS NULL and budget IS NOT NULL "
"AND name = {m_n}  AND year = {m_y};").format(m_n = movieName, m_y = movieYear)
```

As explained above in the MovieProdActors contains the budget and the sum of the salaries of every actor for this movie.

For this query we made the union 3 queries.
The first one is for the basic case if for the given movie has actors that are playing in it and a studio produce the movie
The second one is if the budget is Null in the table it means that in when computing the view, the movie has not been produced by any studio, consequently it's considered to have a crew budget of 0.
The third case is if the movie is produced by a studio but there are no actors that are playing in it, in the view the value for the sum of the salaries is NULL but for this query it's considered as 0 and consequently the query will return budget – 0 = budget

- OverlyinvestedInMovie:

```
query = SQL("SELECT n_roles_actors.cnt * 2 >= n_roles_movie.cnt "
            "FROM "
            "   (SELECT aid, movie_name, movie_year, count(role_played) as cnt "
            "   FROM movieactors GROUP BY aid, movie_name, movie_year) n_roles_actors "
            "JOIN "
            "   (SELECT movie_name, movie_year, count(role_played) as cnt "
            "   FROM movieactors "
            "   GROUP BY movie_name, movie_year) "
            "n_roles_movie On n_roles_movie.movie_name = n_roles_actors.movie_name and n_roles_movie.movie_year = n_roles_actors.movie_year
            "Where n_roles_actors.movie_year = {m_y} and n_roles_actors.movie_name = {m_n} and aid = {aid};").format(
    m_n=movieName, m_y=movieYear, aid=aid)
```

From the table movieActor we grouped the rows by aid, m_name, m_year so that we can use the aggregation function count(role_played) as N_roles_actors.cnt which is the number of roles that the given actor played in the given movie.
From the table movieActor we grouped the rows by m_name and m_year to compute count(role_played) as N_roles_movie.cnt which is the number of roles played in the given movie.

We joined the two tables on the movie attributes (movie_name, movie_year) in the WHERE clause we only take the corresponding movie and actor and in the SELECT clause we checked if the actor played more than half of the roles.

ADVANCED API:

- Franchise revenue:

```
query = SQL("SELECT m.name as m_name, sum(revenue) as movie_revenue "
            "FROM movies m "
            "JOIN productions p on p.movie_name = m.name and p.movie_year = m.year "
            "GROUP BY m.name "
            "UNION "
            " (SELECT m.name as m_name, 0 as movie_revenue"
            " FROM movies m "
            " LEFT JOIN productions p on p.movie_name = m.name and p.movie_year = m.year "
            " GROUP BY m.name "
            "HAVING SUM(revenue) IS NULL )"
            "order by m_name desc ;")
```

This query is composed from the union of to SELECT queries:
The first one is the join between the movies table and the production table on movies' attributes, movie_name and movie_year.
The group by is only on movie name because two movies with the same name but with different year and studio are in the same franchise. And consequently, we use the aggregate function SUM on revenue to sum the revenues of each movie, which are given in the production table
We did an INNER JOIN (JOIN by default) so it will take only the movies that have been produced and don't add to the sum a null value for movies that's haven't been produced.
The second one is based on the same logic, but we used a LEFT OUTTER JOIN because we wanted to see if a franchise have not been produced by any studio and consequently return 0 for this franchise's revenues.

- StudioRevenueByYear

```
query = SQL("SELECT sid, movie_year, sum(revenue)"
            " FROM productions "
            "group by sid, movie_year "
            "order by sid desc, movie_year desc;")
```

This is a basic use of the SUM aggregate functions. The query groups the rows of production by the studio_id (sId) and the release year (movie_year) and sum the revenues of each movie produced during this year

- AverageByGenre:

```
query = SQL("SELECT genre, AVG(age) "
            "FROM ( SELECT distinct a.aid, genre, age "
            "       FROM actorsmoviesnorole as am "
            "       JOIN actors a on am.aid = a.aid "
            "       JOIN movies m on movie_year = m.year and movie_name = m.name) B "
            "group by genre "
            "ORDER BY genre;")
```

In the B subquery joined the view actorsmovienorole and the table actors to also get the age of the actor, and we joined this with the table movies to also get the genre of the movie.
From this we select the Id of the actor aId, the genre of the movie and the age of the actor, we used distinct key word to avoid the duplicates from an actor played that in more than one movie.
We grouped the rows of this subquery by genre use the AVG aggregation function to get the average age of the actors for each genre
Finally, we select the genre and the average of the ages to return a list of tuples.

- getFanCritics:

```
query = SQL("select cid, A.sid "
            "FROM "
            "   (SELECT sid, count(*) as prod_cnt "
            "   FROM productions p "
            "   GROUP BY sid) A "
            "JOIN "
            "   (SELECT cid, sid, count(*) as critic_on_studio_cnt "
            "   FROM moviecritics mc "
            "   JOIN productions p on mc.movie_year = p.movie_year and mc.movie_name = p.movie_name "
            "   GROUP BY cid, sid) B ON A.sid = B.sid "
            "WHERE A.prod_cnt = B.critic_on_studio_cnt"
            " ORDER BY cid DESC, sid DESC;")
```

As we can see there are several sub-queries in this query.

The A subquery is a basic COUNT of the movies that a studio produced in the production table.

The B subquery gives for each pair of studios (sId) in production and critics (cId) in moviecritcs the number of movies from the studio sId that the critic cId has rated.

By making a join on the two subqueries A and B on the studio's id (sId) we can get for pair (sId, cId) the number of movies produced by sId on one side and on the other the number of rating that cId made on movies of sId. And consequently, we can compare them in the WHERE close to return the relevant critics and studios

- getExclusiveActors

```
query = SQL("SELECT distinct aid, sid "
            "FROM actorStudio "
            "WHERE aid IN "
            "   (SELECT aid "
            "   FROM actorStudio "
            "   GROUP BY aid "
            "   having count(distinct sid) = 1) "
            "ORDER BY aid DESC;")
```

As we remember in if an actor aId plays in a movie that the studio sId produced, (aId, sId) will be in the view actorStudio

So in the sub-query we have the number the id of the actors that play for movies of only one studio (distinct keyword is here because if an actor play in two movies of the same studio it will be registered two times in the actorStudio view)

So if in the WHERE close we select only the actors that are in the results of the subquery we will get only exclusive actors ids and their studio