Assignment 3 – Database Systems

Database schema structure:

Table 1: **title(id, name, year, minutes, ratings)**

id – a unique identifier of the movie.

name – the name of the movie.

year – the year the production of the movie started.

minutes – the duration of the movie in minutes.

ratings – average imdb rating of the movie, movies with no rating will be counted as having their average rating equal to zero.

(id) is the table primary key.

Table 2: **genre(id, genre)**

id – the id of the movie with that genre.

genre – genre of the movie.

(id, genre) is the table primary key.

(id) is a foreign key references title(id).

Table 3: **person(id, name)**

id – a unique identifier of the person.

name – the name of the person.

(id) is the table primary key.

Table 4: **title_person(title_id, person_id, job)**

title_id – id of a movie.

person_id – id of a person.

job – job of the person on the movie's production.

(title_id, person_id, job) is the table primary key.

(title_id) is a foreign key references title(id).

(person_id) is a foreign key references person(id).

Table 5: **profession(id, profession)**

person_id – id of a person.

profession – profession of the person.

(id, profession) is the table primary key.

(id) is a foreign key references person(id).

Database design:

In the web application we can search for a movie by genre. We could have added a column of genres to title, but because any movie can have multiple genres and each genre can relate to multiple movies – we have a many to many relationships. That means that we will have to duplicate each row in title by the number of genres the movie has. To solve that we made a different table for genres and added a foreign key that references title.id .

We also want to search by profession, so the previous problem happens again with person and profession. We solved it by making a new table of professions.

On the contrary, each movie can have only one average rating, therefore we merged column averageRating from title.ratings in the title table.

The person_title table is used is for connecting between the person table and the title table.

Optimizations:

The original "ids" of title and person have 7-10 digits and because the data from our source came with other types of media, such as TV shows, TV episodes, games, etc... we filtered out a lot of irrelevant data. Therefore, we get a much smaller dataset and to save space we chose to change the id of each title and person (in every place they are in) to mediunint. This process takes an additional time, but this prosses happens only once in the data retrieval process and it may help shorten the time in the execution of the five main queries (the shorter ids lead to shorter tuples, and that leads to the possibility of loading more tuples into pages and less I/O).

We've also added a sorted index (a B-tree) on title.year, title_person.person_id and full-text indices on title.name, genre.genre, person.name.

The queries:

query_1: Gets a genre and two year numbers, and returns a table that has movie genres, movie ids, movie name, years, duration in minutes and average imdb ratings columns (in that order) that contains only the information of titles of the given genre that started the production between the given years (including the boundaries.).

The result will be sorted by descending average rating.

To search the genre in genre.genre we've added a full-text index on genre.genre, to search by year we've added a B-tree index on title.year (we've noticed that we have to add both because for large enough ranges it'll be more efficient to search by genre.genre first and for smaller year

ranges it'll be more efficient to search first by title.year.), no extra indices are needed because the indices that come with the primary keys suffice.

query_2: A full-text query that searches by movie name, and returns the matching rows in the title table.

The full-text index on title.name optimizes this query.

query_3: Searches by movie name (via match() ... against in natural language mode) and returns a table where the first column contains the ids of the movies, the second their names, a type of in this movie's production, and how many people did this job.

The result will be sorted by ascending title id.

There's a full-text index on title.name for this query, no-more is needed because the indices that come with the primary keys are enough for the join between the title and title_person tables preformed by this query.

query_4: Get for a person the average rating of average imdb movie ratings where the person was part of the production, by the person's name.
(Movies with no rating will be counted as having their average rating equal to zero.)

The result will be sorted by descending average of averages.

The query joins the title_person, person and title tables, based on title id and person id.
We've added a full-text index on person.name for this query, we've also added a B-tree index for title_person.person_id for this query because here the index that comes with the primary key does not optimize! It's a (title_id, person_id, job) index and therefore it cannot help with searching by a person_id but the person primary key index suffices for joining the title table.

query_5: Gets all industry professions having amount of people working in them greater than the number given.

The query joins the person and profession tables based on ids.
Here the indices that come with the primary keys are enough for the query.

Outline:

We took the data from https://developer.imdb.com/non-commercial-datasets/

Those data sets have more than one million tuples each and when we tried to load the first table, it took over an hour. Therefore, we took only the first 50,000 tuples of each table.

Those data sets contain a lot of data we do not need. hence, we filtered out any unnecessary data. We filtered out tuples with nulls, adult's movies, TV shows, games... we did not add columns like birth year of a person, death year of a person, original title of a title...

When we created the tables, we added temporary columns and temporary foreign keys so we can later change the ids of the films and people like we mentioned in the optimization section and filter unnecessary data by foreign keys.

We added permanent auto increment unique ids as primary keys to person and title so we can switch them with the previous ones.

First, we loaded the titles filtered by type of movie.

Then, we split the genres of the titles and assigned them their title id.

We took the average rating from title.ratings and updated the title table by assigning each rating to the correct title. If the title did not have a rating, then it is set to 0.

We loaded into the genre table the genres we split before and added only the ones who connected to the title tables (the id of title without a genre will not be in this table).

Next, we loaded into person table.

Then, we split the profession of the people and assigned them their people id.

After that, we loaded into title_person only tuples which have the id of a film from the title table and the id of a person from the person table.

Later, we loaded into the profession table the professions we split before and added only the ones who connected to the person tables (the id of person without a profession will not be in this table).

This sums up the first part of the data retrieval. Next, we filtered out more unnecessary data, we set the new ids, added primary keys, added foreign keys, removed keys, and dropped columns.

We deleted tuples from profession and person where the person is not in title_person table.

After that, we fixed profession.id, add new keys and dropped the previous foreign key and the temp column.

Then, we fixed title_person.title_id and title_person.person_id, added new keys and dropped the temp1 and temp2 columns and the foreign keys related to those columns.

Then, we dropped the temp column from person.

Next, we fixed genre.id, added new keys and dropped the genre.temp column and its foreign key.

Finaly, we dropped from title the temp, type and adult columns.

We used ipynb file for the queries execution.