Assignment Number 3 Normalization
-Deepika Kini

Note: final sql file name with codes for question 1:
Final python file for question:

Q. 1.

The joins are implemented in sql using the tables made in homework 3:

To remove the actors with more than one characters in a movie, I've created a new table using the query:

```
create table one_character_per_movie as
select title, actor, count(character)
from atc
group by title, actor
having count(character) = 1
```

The main query to join all the tables involved is:

```
create table final_norm_grouped as
SELECT title.id,title.type,title.startYear, title.runtime, title.avgrating, genre.id as genre_id,
genre.genre, atc.actor, member.birthyear, character.character
from atc
join character on atc.character = character.id
join actor on actor.actor = atc.actor
join title on actor.title = title.id and title.id = atc.title
join title_genre on title.id = title_genre.title
join genre on title_genre.genre = genre.id
join member on actor.actor = member.id
where title.runtime>=90
and concat(title.id,member.id)  not in (select concat(title, actor) in
from atc
group by title, actor
having count(character) >1)
limit 1000000;
```

I was getting the error of "space running out on disk" or "postgres disconnected from server" and had to limit the rows to 1 million

The final table name is : **final_norm_grouped**
The excerpt from table is saved in csv file: data-1677166782525.csv

I have renamed these columns in python
—-

Made a primary_key column that concats the columns involved to make this table unique using the following commands: (it isn't used for further computation)

```
ALTER TABLE final_norm_grouped ADD column primary_key varchar(50);
update final_norm_grouped set primary_key = concat(id , genre_id,actor);
ALTER TABLE final_norm_grouped ADD PRIMARY KEY (primary_key);
```

(referred to https://stackoverflow.com/questions/38732376/how-do-i-add-a-concatenated-column-to-existing-table since wasn't able to directly make the key, instead the technique is to initialize the variable and then assign value which in this case is concat some columns)

Q.2

Taking the data into consideration and trying to have a clear head (since I already have reviewed and worked on the data), functional dependencies basically are column(s) associateions wherein similar tuple for one column leads to same tuple values for the other column for all tuple cases.
trying to understand the IMDB dataset, I would provide the following functional dependencies of the dataset(bolded) and some comments:

movieID - > title and **title - > movieID** (should be the other way around as well)
(just an example for showing bidirectional FD)
movieID - > type
movieID - > startYear (the other way doesn't hold since many movies can be released in the year 1990)
movieID - > avgrating (1:1 table mapped previously where movieID was the PK)
movieID - > runtime

movie ID not FD on genre since movieID can be associated with many genres
But **genreID -> genre** since there is a 1:1 mapping and vice versa: **genre - > genreID**

Same goes with movieID and MemberID since it's a M:N mapping.(this extends to all columns having FD on movieID)
Actor will have the same birthdate for all rows and hence **actor - > birthYear** but the opposite isn't true
Actor to character logically doesn't show an FD since an actor can have many roles. But since the data extracted by us has only one character per movie, we can say that **movieID, actorID -> character**

Considering other FDs where there are two columns on the left:
All combinations of similar left side values can be added to left side:
Eg: **movieID, type, startYear…->runtime…**
Other combinations of candidate key (Also movieID and genreID/memberID distinguish the rows of type, startYear, avgrating, runtime):
movieID, genreID, memberID - >type, startYear, avgrating, runtime
(not normalized)

End Goal of the assignment:
The key here is the movieID, genreID(one movie can have multiple genres), MemberID (M:N relation with movieID)
Since there is no primary key with one column, there will be redundancy of data which is determined by the FDs, and can be decomposed/split into 2nd/3rd NF for better database architecture

---

Q. 3.

For 10,000 rows:
The trivial ones are not considered (movieID - > movieID).
Only 1 column on the left side is considered
Output:
movieId->type, startYear, runtime, avgRating,
type->
startYear->
runtime->
avgRating->
genreId->genre,
genre->genreId,
memberId->birthYear, ~~type (comes in when 1000 rows are considered)~~
birthYear->
character->

7

The code checks all combinations in the table, within which it checks if the values of rows for one column are same for the corresponding same valued rows in the other columns.

It breaks if this isn't true and continues with another combination. If it gets through all rows, without breaking, it means the column combination are seen a functional dependency.

Runs for more than 3 hours for a dataset of 1 million rows.
If taking only 10000 rows, the outcome is what is seen above (takes 1.75 hours approx)

Q.4.
The outcome is the same for manual (canonical & non trivial)vs naive approach

If we consider multiple character values for one title-member pair (knowing the extent of data), Q.2 would not include a functional dependency memberID, movieID->character

We're also considering genreID as a prime attribute instead of genre (although it too forms a candidate key pair with movieID and memberID). For better practice, we use genreID as candidate key, hence not giving weightage to genre - > genreID.

Q.5.

The dataset is already in 1NF (since we made genre and character into atomic attributes by making tables of their own and associating them with a key for connecting them with the main tables)

2NF: partial dependencies based on all subsets of candidate key:
The columns avgrating, type, startYear, runtime are not only related to movieId but also genreID and MemberID which can be shown as below:
MovieID, GenreID, MemberID ->avgrating, type, startYear, runtime
Character is dependent on MovieID and MemberID which can also be added to the above list:
MovieID->avgrating, type, startYear, runtime

While genre is dependent on genreID , birthYear is dependent on MemberID:
genreID -> genre
memberID->birthDate
MovieID, memberID -> character

3NF:
no transitive dependency. Hence it is already in 3NF

Candidate key set = (MovieID, MemberID, GenreID)

We are removing the extraneous attributes such as MovieID, runtime - > avgrating… and keeping α (left side set of columns) unique
Hence the **Canonical cover** will be the table wise relations
Fc= {movieID -> avgrating, type, startYear, runtime, MemberID -> birthYear, GenreID -> genre, MovieID, MemberID ->character}
Final decomposition Table schemas:(star schema)
Fact table = (MovieID, MemberID, GenreID)
Title table = (*MovieID*, avgrating, type, startYear, runtime)
Member table = (*MemberID*, birthYear)
Genre table = (*GenreID*, genre)
Character table = (MovieID, MemberID, character)

Underlined columns are primary keys and italicized are foreign keys

—----------x—----------