Homework 3

Dongxu Han

1. At first, my program uses following sql to create table and set the auto-increment id.

   ```
   drop table if exists new_relation;

   create table new_relation(
       movieId int, type varchar(50),
       startYear smallint,
       runtime int,
       avgRating numeric(3, 1),
       genreId int,
       genre varchar(50),
       memberId int,
       birthyear smallint,
       role int
   );
   alter table new_relation primary key (movieid,genreid,
       memberid);


   alter table new_relation
   foreign key (genreid) references movie_genre(genre);


   alter table new_relation
   foreign key (memberid) references member(id);


   alter table new_relation
   foreign key (movieid) references movie(id);
   ```

   Then using following sql to insert data, which is a join query of tables, to new_relation.

   ```
   insert into new_relation
   select mv.id,
          mv.type,
          mv.startYear,
          mv.runtime,
          mv.avgRating,
          gr.id,
          gr.name,
          mb.id,
          mb.birthyear,
          amr.role
   from
    (select id,
            type,
   ```

```
         startYear,
         runtime,
         avgRating
  from movie
  where runtime > 90) as mv
join (select actor, movie, role
  from actor_movie_role as inamr
  where not exists
  (select movie,
            actor
     from actor_movie_role
     where movie = inamr.movie
     group by movie,
              actor having count(role) > 1)) as amr on amr.
                 movie = mv.id
join movie_genre as mg on mv.id = mg.movie
join genre as gr on gr.id = mg.genre
join
 (select id,
         birthyear
  from member) as mb on mb.id = amr.actor;
```

Now let's see this long SQL script.
The first sub-query is selecting movies whose runtime is greater than 90 minutes. Since we are joining multiple tables, use constrains on single table could help to reduce task of joins.

```
select id,
         type,
         startYear,
         runtime,
         avgRating
  from movie
  where runtime > 90
```

The second sub-query is selecting movies in which actors act only one role.

```
select movie,
       actor
from actor_movie_role
where movie = inamr.movie
group by movie,
         actor having count(role) > 1
```

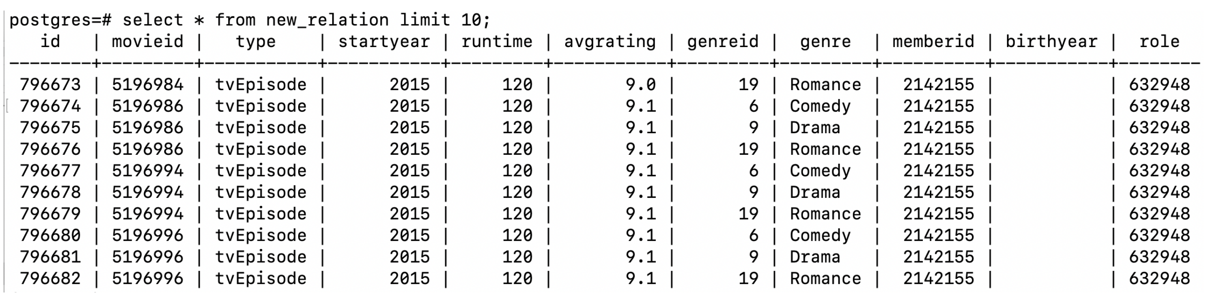
After knowing the actor and movie, we need the role information. So we have an query after above to query all data.

```
select actor,
       movie,
       role
from actor_movie_role as inamr
where not exists
  (select movie,
          actor
   from actor_movie_role
   where movie = inamr.movie
   group by movie,
            actor having count(role) > 1)
```

And then we join it with movie to get other informations

```
select id,
       type,
       startYear,
       runtime,
       avgRating
from movie
where runtime > 90) as mv
 join
  (select actor,
          movie,
          role
   from actor_movie_role as inamr
   where not exists
     (select movie,
             actor
      from actor_movie_role
      where movie = inamr.movie
      group by movie,
               actor having count(role) > 1)) as amr on amr.
                   movie = mv.id
```

For other attributes, we get them by simply join.
1060624 tuples are inserted. The final execution result and a few tuples are shown below.



```
postgres=# select * from new_relation limit 10;
   id   | movieid |   type    | startyear | runtime | avgrating | genreid |  genre  | memberid | birthyear |  role
--------+---------+-----------+-----------+---------+-----------+---------+---------+----------+-----------+--------
 796673 | 5196984 | tvEpisode |      2015 |     120 |       9.0 |      19 | Romance |  2142155 |           | 632948
 796674 | 5196986 | tvEpisode |      2015 |     120 |       9.1 |       6 | Comedy  |  2142155 |           | 632948
 796675 | 5196986 | tvEpisode |      2015 |     120 |       9.1 |       9 | Drama   |  2142155 |           | 632948
 796676 | 5196986 | tvEpisode |      2015 |     120 |       9.1 |      19 | Romance |  2142155 |           | 632948
 796677 | 5196994 | tvEpisode |      2015 |     120 |       9.1 |       6 | Comedy  |  2142155 |           | 632948
 796678 | 5196994 | tvEpisode |      2015 |     120 |       9.1 |       9 | Drama   |  2142155 |           | 632948
 796679 | 5196994 | tvEpisode |      2015 |     120 |       9.1 |      19 | Romance |  2142155 |           | 632948
 796680 | 5196996 | tvEpisode |      2015 |     120 |       9.1 |       6 | Comedy  |  2142155 |           | 632948
 796681 | 5196996 | tvEpisode |      2015 |     120 |       9.1 |       9 | Drama   |  2142155 |           | 632948
 796682 | 5196996 | tvEpisode |      2015 |     120 |       9.1 |      19 | Romance |  2142155 |           | 632948
```

2. The program is shown in Program2.java.
   In a naive approach, I firstly get all combinations of all attributes of the relation. Then I check if the combination -¿ an attribute. The relation has 10 attributes and so I get $\approx 2^10$ combinations. For each attribute, I should check if it depends on some combinations. So $\approx 2^10 \times 10$ times of checking. The main part of checking the functional dependency is the following SQL:

   ```
   select combination
   from new_relation
   group by combination
   having count(attribute) = 1;
   ```

   In the SQL script, we are checking if *combination* $\rightarrow$ *attribute* holds. Considering the definition of functional dependency, for any two tuples, t1, t2,

in the relation, if $t1[combination] = t2[combination]$ and $t1[attribute] = t2[attribute]$, then $combination \rightarrow attribute$ holds. In the query, we add constrain of "count(distinct attribute) = 1" on "group by combination". This means for each value of combination, there exists only one value of attribute. Then the dependency exists.
In the program, I only check those dependencies with one attribute on the right side, because $a \rightarrow b, a \rightarrow c$ is equivalent to $a \rightarrow bc$. So just focus on the left side.
To improve the performance, for each attribute, I create a single thread to check functional dependency between it and combinations. So 11 threads are needed.
The result of execution of the program is shown below. It costs about 2.4 hours.
The output of program:

```
Program started.
Retrieve attributes from relation: success.
Generate combinations: success.
Created threads to find functional dependencies.

Program finish in 8,336 seconds

9248 functional dependencies are found in naive approach.
Process finished with exit code 0
```

3. My Program3 is modified based on Program2. Since we have constrains on size of the left-hand side, it only get combinations with one or two attributes on left-hand side. Then we use the same query to try to determine if $t1[left] = t2[left]$ and $t1[right] = t2[right]$.
   The main idea is traversing the lattice. Since we cannot do anything about level 0 and level 1, I start the main detection from level 2. The main parts of Program3 is dependency computation and pruning.
   The dependency computation algorithm is shown below:

```
for each $X \in L_\ell$ do
    $\mathcal{C}^+(X) := \bigcap_{A \in X} \mathcal{C}^+(X \setminus \{A\})$
for each $X \in L_\ell$ do
    for each $A \in X \cap \mathcal{C}^+(X)$ do
        if $X \setminus \{A\} \rightarrow A$ is valid then
            output $X \setminus \{A\} \rightarrow A$
            remove $A$ from $\mathcal{C}^+(X)$
            remove all $B$ in $R \setminus X$ from $\mathcal{C}^+(X)$
```

The pruning algorithm is shown below:

```
for each $X \in L_\ell$ do
    if $\mathcal{C}^+(X) = \emptyset$ do
        delete $X$ from $L_\ell$
```

The output of Program3:

```
Program started.
Retrieve attributes from relation: success.
[movieid] → type
[movieid] → avgrating
[movieid] → startyear
[movieid] → runtime
[memberid] → birthyear
[genre] → genreid
[genreid] → genre
7 functional dependencies are found in pruning approach.

Program finish in 86 seconds

Process finished with exit code 0
```

4. If we don't restrict that an actor only plays a single role in a given movie, then we only have restriction on the runtime of movie, which is "more than 90". Then we would include those movie, in which a single actor could plays more than one role. Since we use "movieid", "memberid", "genreid" as primary key of the relation, this would be affected with duplicate key error since in a given movie of a given genre for a given actor(member) there exists more than one roles. So the primary key would be"movieid", "memberid", "genreid", "role". Besides this, we would have more redundant data due to join of multiple genre join multiple actors. The functional dependencies we got in question 3 would change cause the role would be part of candidate key, which previously depends on "movieid", "memberid" due to the restriction of "one actor acts one role in one movie".

5.