CSCI-620 – INTRODUCTION TO BIG DATA

1. For creating this relation, I performed 6 inner joins altogether. Each of my queries perform two inner joins and I used a total of 3 queries to join all necessary tables in to a single relation with the attributes as listed in the question.

   The first of the inner join in the first query merges the two tables movie and movie_genre into one table with movie and genre assigned the attribute movieId and genreId respectively in the newly formed table. The join is performed on the movieId which is common to both the movie table and movie_genre table. Then, the second join is performed on the genre table and the newly formed relation obtained by joining movie and movie_genre. The join is performed based on the genre. Thus, I obtain the relation where I have movieId, type, startYear, runtime, avgRating, genreId, genre. I call this intermediate relation as movie_genre_relation. Both the joins are performed in the same query with restriction that the runtime attribute should have valued greater than equal to 90.

   Similarly, I perform the joins on the relation I got from the previous query i.e. movie_genre_relation and the table movie_actor based on the movieId. I, then use the resultant relation to join member table based the actor attribute which I call the memberId attribute. This relation will have the attributes movieId, type, startYear, runtime, avgRating, genreId, genre, memberId, birthyear. I call this relation movie_genre_actor.

   From this relation, I join with the table actor_movie_role on the attributes movieId and memberId. This will result in the final relation, movie_genre_actor_role with the following attributes: movieId, type, startYear, runtime, avgRating, genreId, genre, memberId, birthyear and role. I, then delete the rows from table where an actor has played more than two roles from in the same movie. This is done using the Group by clause, the tuples are grouped based on movieId, memberId and genreId, the we check whether any of the groups have a count of actors greater than 1. If so, we delete them because if for the same movie with the same genre, a memberId repeats again, then the member has performed two different roles in the movie which violated the constraint. Since a combination of movieId, genreId and memberId can uniquely determine every other attribute in the table, we can set movieId, genreId and memberId as the primary key.

2. The naïve approach is used to find the all the possible functional dependencies which includes trivial dependencies. This method gives the left-hand side of the dependency as any super key (super set of keys) that can determine an attribute on it owns, thus contains many trivial dependencies. Thus, it does not give the minimal canonical cover where the left-hand side is just a candidate key.

   In my program, we have a list of all the 10 attributes which we later use as the Right-hand side element for checking functional dependencies. The naïve approach iterates through all the 10 attributes in the list and forms multiple non-repetitive combinations of the attributes and stores them in another array. We then loop through this array where each of the elements of the array will be used as the Left-Hand side for each of the functional dependency. We then use, an inner loop to iterate through the list containing only the 10 attributes. We use each of the elements of this list as the right-hand side. We then check for the functional dependency using a query inside the inner loop with the respective LHS and RHS elements from their respective list. The query will return an empty ResultSet if a functional dependency exists. Thus, we whenever we get a functional dependency, we count it.

   Execution Time: 90 minutes.

   The execution time is long because the program counts in trivial dependencies too. There was a total of 205 functional dependencies which includes both the trivial and non-trivial dependencies.

3. I have hardcoded the first level as the lattice as the individual columns. I use a function to find multiple combinations which are the next level of the lattice. I use hashmap as the internal data structure. The key of every hashmap is a list of attributes that can be functionally determined by the key. I program makes sure to prune trivial attributes through conditional checks. Only non-trivial dependencies are added to the hashmap. If a trivial dependency is about to added, the program checks if a subset of the LHS in the dependency already exists in the hashmap and if it does, the dependency doesn't get added to the map. Thus, in the end we get minimal functional dependency.

   Example of Pruning: If $\boldsymbol{movieId} \rightarrow \boldsymbol{runtime}$ already exists,
   And the LHS now is $\boldsymbol{movieId, avgRating} \rightarrow \boldsymbol{runtime.}$
   Then we don't add this LHS because, we already have a non-trivial dependency $\boldsymbol{movieId} \rightarrow \boldsymbol{runtime}$. So therefore**,** $\boldsymbol{movieId, avgRating} \rightarrow \boldsymbol{runtime}$ is a trivial dependency and doesn't get added.

4. The minimal dependencies/canonical cover from question 3 are as follows:

$$\boldsymbol{movieId \rightarrow runtime}$$
$$\boldsymbol{movieId \rightarrow avgRating}$$
$$\boldsymbol{movieId \rightarrow startYear}$$
$$\boldsymbol{runtime \rightarrow type}$$
$$\boldsymbol{genreId \rightarrow genre}$$
$$\boldsymbol{genre \rightarrow genreId}$$
$$\boldsymbol{memberId \rightarrow birthYear}$$
$$\boldsymbol{movieId, memberId \rightarrow role}$$

Primary key for the relation: {movieId, memberId, genreId}

From the above functional dependencies, it clear that when we don't restrict ourselves to actors who played only one role in a movie, the functional dependency, $\boldsymbol{movieId, memberId \rightarrow role,}$ will no longer hold as for the same (movieId, memberId), the roles may vary. Also, since the functional dependency is already in it's minimal form for Role, therefore every other key will only be super set of {movieId, memberId}.

Therefore, from the existing attributes we cannot form any candidate key that can functionally determine role. To create a functional dependency, we must add another attribute that can functionally determine role which can be done by adding an auto-incremented primary key for every row.

5. Steps needed to compute 3NF decomposition:
Initial Relation,
R: **{movieId, type, startYear, runtime, avgRating, genreId, genre, memberId, birthyear, role};**
Primary Key, PK: {movieId, memberId, genreId}
(1) Eliminate redundant FDs, resulting in a canonical cover Fc of F:

The functional dependencies from question 3 are already in canonical from and therefore cannot be reduced further. The canonical covers are as follows:

$$\boldsymbol{movieId \rightarrow runtime}$$
$$\boldsymbol{movieId \rightarrow avgRating}$$
$$\boldsymbol{movieId \rightarrow startYear}$$
$$\boldsymbol{runtime \rightarrow type}$$
$$\boldsymbol{genreId \rightarrow genre}$$
$$\boldsymbol{genre \rightarrow genreId}$$
$$\boldsymbol{memberId \rightarrow birthYear}$$
$$\boldsymbol{movieId, memberId \rightarrow role}$$

(2) Create a relation Ri = XY for each FD X → Y in Fc.

a. Since $\boldsymbol{movieId \rightarrow runtime}$ and $\boldsymbol{runtime \rightarrow type,}$ since there is a transitive dependency, we can create a separate relation,

   R1 = {runtime, type} and we can remove **type** from R.

   PK = {runtime}

   Therefore now, **R**: **{movieId, startYear, runtime, avgRating, genreId, genre, memberId, birthyear, role}.**

b. Use the Armstrong's union rule, to combine functional dependencies with only movieId as the LHS. We get:
   $\boldsymbol{movieId \rightarrow runtime, avgRating, startYear}$
   Therefore, **R_2 = {movieId, runtime, avgRating, startYear};**
   PK = {movieId};

c. Now create another relation for, $\boldsymbol{genreId \rightarrow genre}$ and $\boldsymbol{memberId \rightarrow birthYear}$ **and** $\boldsymbol{movieId, memberId \rightarrow role}$

   R_3 = {genreId, genre}; PK = {genreId};

   R_4 = {memberId, birthYear}; PK = {memberId};

   R_5 = {movieId, memberId, role}; PK = {movieId, memberId};

(3) If the key K of R does not occur in any relation Ri, create one more relation Ri=K:

Since the key of R_2, R_3, R_4 does not occur in any other relation, create a relation with all these keys:
R_6 = {movieId, genreId, memberId} = Candidate Key for Relation R.