3.

In order to find the invalid Movie\_Actor relationships with respect to roles, we only need to find the null-value role in Actor\_Movie\_Role table.

First, let Movie table join Movie\_Genre table when id in Movie table is equal to movie in Movie\_Genre table. Then let it join Member table when id in Member table is equal to movie in Movie\_Genre table. Last, select the name when deathYear is ‘’, name is begin with ‘Phi’ and startYear is not ‘2014’.

First, we have to let Movie table join Movie-Genre table when id in Movie table is equal to movie in Movie\_Genre table. Then let it join Genre table when id in Genre table is equal to genra in the last table. Then let it join Movie\_Producer table when movie in Movie\_Producer table is equal to id in last table, which can filter all producer. Then let it join Member table when id in Member table is equal to producer in last table. Last, select the name when startYear is ‘2017’ and genre which conclude ‘Talk-Show’. Now, we have all the eligible name, then find the name which appears most often.

Next, for the second query, we need to let Member table join Movie\_Producer table, which can filter all producers. Then we select the name which contains ‘Gill’.

First, let Member table join Movie\_Producer table when id in Member table equals producer in Movie\_Prodcer table. Then let Movie table join it when id in Movie table equals movie in last table. Then select the name whose deathYear is ‘’ and runtime grater than 120. Now, we only need to find the name which appears most often.

First, let Member table join Actor\_Movie\_Role table when id in Member table equals actor in Actor\_Movie\_Role table. Then let Role table join it when id in Role table equals role in last table. Then select the name whose deathYear is ‘’, role include ‘Jesus’ and ‘Christ’.

4.

role is null(Actor\_Movie\_Role)

mem.name (deathYear=’’ ^ name LIKE ‘Phi%’ ^ startYear IS DISTINCT FROM ‘2014’(mem.id=actor(mov.id=movie(Movie X Movie\_Genre) X Member)))

mem.name(startYear=’2017’ ^ gen.genra LIKE ‘%Talk-Show%’ (mem.id =iproducer (pro.movie=id (gen.id=genre(mov.id=movie(Movie X Movie\_Genre) X Genre) X Movie\_Producer) X Member)))

mem.name (name LIKE ‘%Gill%’ (mov.id=movie(Member X Movie\_Producer) ))

mem.name (deathYear=’’ ^ to\_number(runtime,9999999999999999999)>120(mov.id=movie(mem.id=producer(Member X Movie\_Producer) X Movie)))

mem.name (deathYear=’’ ^ role LIKE ‘%Jesus%’ ^ role LIKE ‘%Christ%’ (ro.id=jar.role(mem.id=actor(Member X Actor\_Movie\_Role) X Role)))

5.

When I retrieve the data. I used id, startYear and runtime from Movie table; id and genre from Genre table; movie from Movie\_Genre table; id, name and deathYear from Member table; actor and movie from Movie\_Actor table; producer and movie from Movie\_Producer table; id and role from Role table; actor and role from Actor\_Movie\_Role. So I just create indexes for these column.

When postgres executes a sql statement, the default way is to scan the whole table according to query and add it to the result set if the data matches the query. If we add an index to a field, the query will first locate the row of specific value in the index list, greatly reducing the time of matching rows. Thus, significantly increasing the speed of the query.