# Homework2 report

Dongxu Han

1.

In my program, I started multiple threads to load data into database. Even though I disable the foreign key between tables, tables which have attributes referencing to attributes in member or movie table would be loaded after movie or member. This the logical order. There are several problem I found and solved.

a) How to deal with genre?

Since there are not much data about genre, I store it in a map, whose key is genre and value is id. This is going to be faster than insert it into database with operation to deal with conflict.

b) How to deal with duplicate data while loading data into movie_actor, relation table like this?

I find it useful to use “on conflict do nothing” to ignore the last operation due to the data has already existed. This is going to be faster than querying it before inserting.

*The output of loading data:*
*Thread loads member within 5.99 minutes*
*Thread loads movie, genre, movie_genre within 11.02 minutes.*
*Thread update ratings within 1.55 minutes.*
*Thread loads write and direct within 28.71 minutes.*
*Thread loads movie_actor, movie_producer, role, actor_movie_role within 57.87 minutes.*
*Finish loading all data within 57.87 minutes.*

This is reasonable because we do operations on four tables in the thread loading data with about one hour.

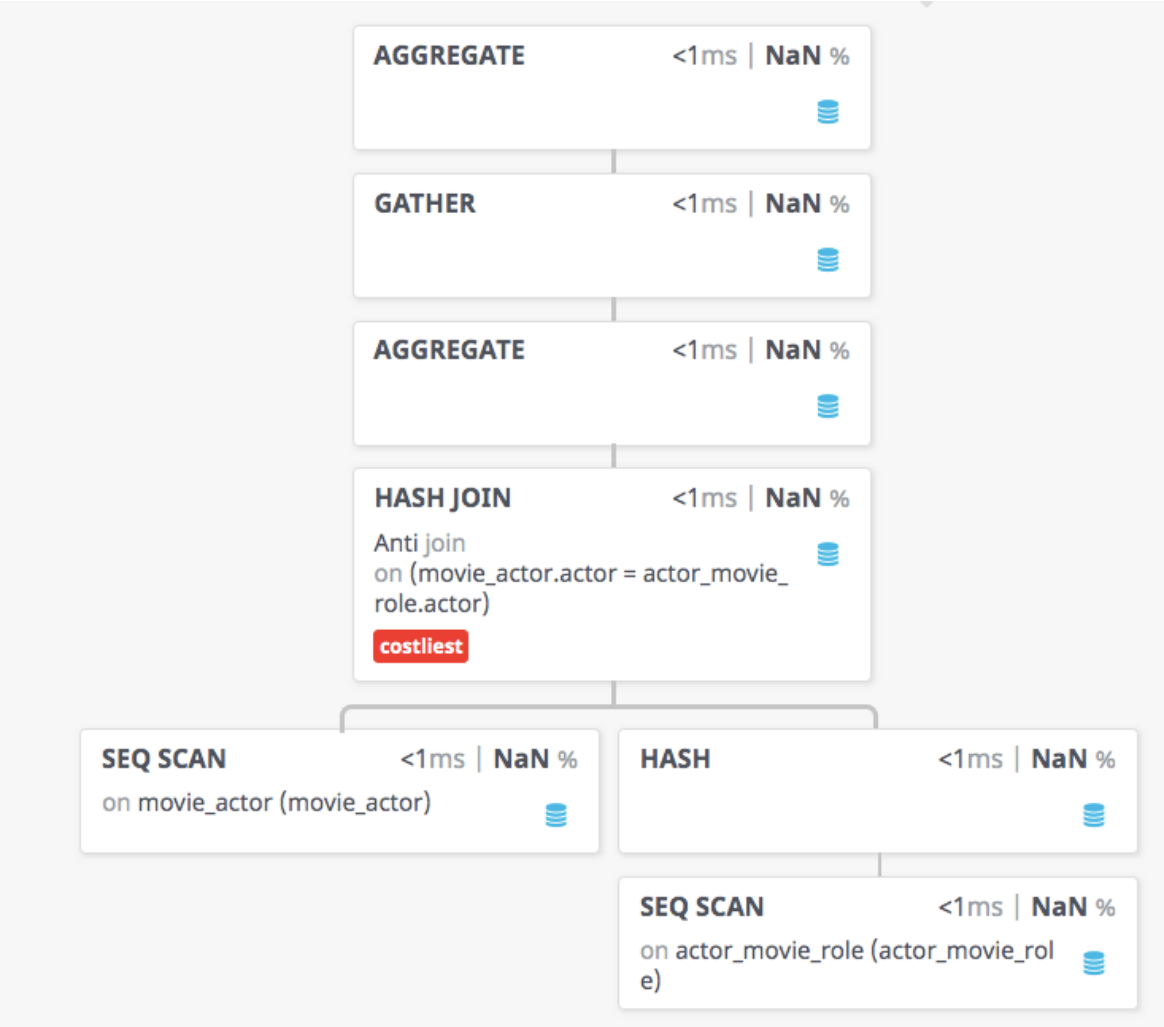
2 & 3.

2.1.

```
select count(*)
from movie_actor
where not exists (select * from actor_movie_role
where movie_actor.actor = actor_movie_role.actor);
```

Output of program:
*Finish retrieve number of invalid Movie_Actor relationships with respect to roles*
*in **54.12** seconds*
*Count: 465025*

In this query, we do join two tables, movie_actor and actor_movie_role. Each time the server deals with one row, it will do subquery in the parenthesis. If the subquery does return any result, then it means the actor does play some roles, whatever, in some movie. Then this would be counted as 1. So at last, we got number of actors who play roles in some movies. As shown in the picture, the join would be the costliest part.

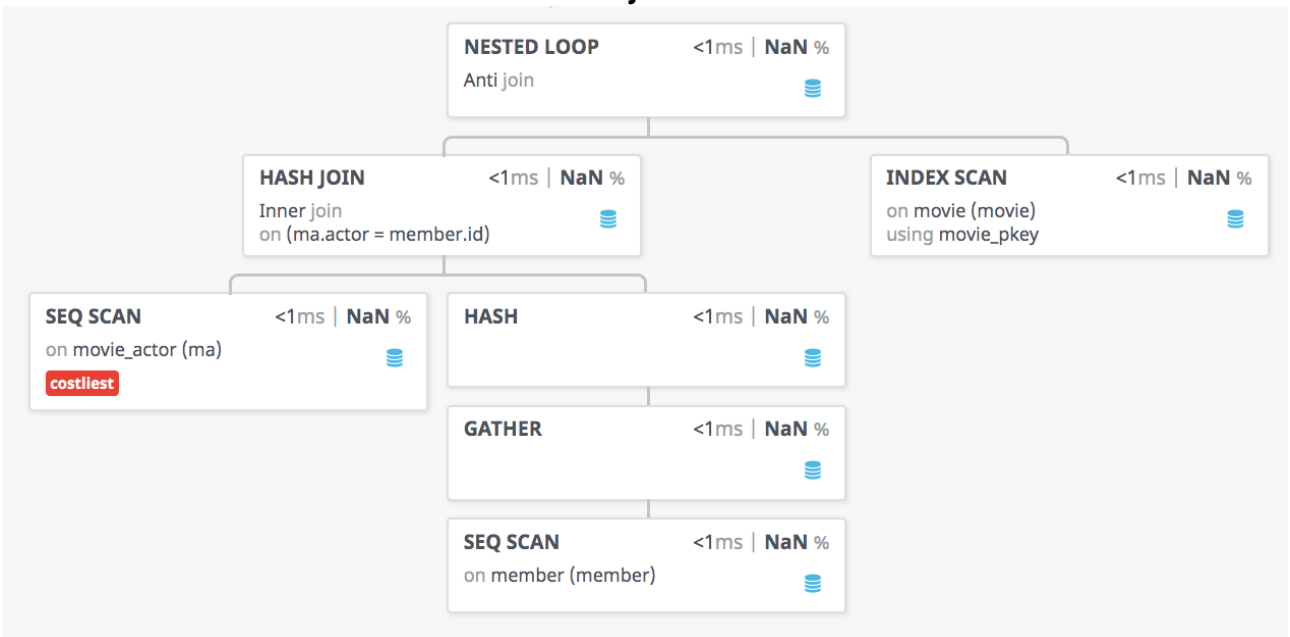



2.2.
select mb.name
from movie_actor as ma

```
join (select id, name from member where deathYear is null and
name like 'Phi%') as mb
on ma.actor = mb.id
where not exists(select * from movie where startYear = 2014 and
ma.movie = movie.id);
```

Output of program:
*Finish retrieve alive actors whose name starts with "Phi" and did not participate in any movie in 2014*
*in **15.45** seconds*
*number of results: 4165*
*Few examples are shown below.*
*Phi Bulani*
*Phi Clarke*
*Phi Hung Nguyen*
*Phi Huynh*
*Phi Lan*

For this query, this exclude those movie in 2014 in our subquery. Then we join the result of another subquery, which query alive person whose name starts with "Phi", with movie_actor table. In this way we got alive person who named "Phi…" and doesn't act in any movie in 2014. We can also see in the picture, we have two join and the costliest part is on movie_actor since we scan this table to exclude actor in 2014 movie and join it with member table.



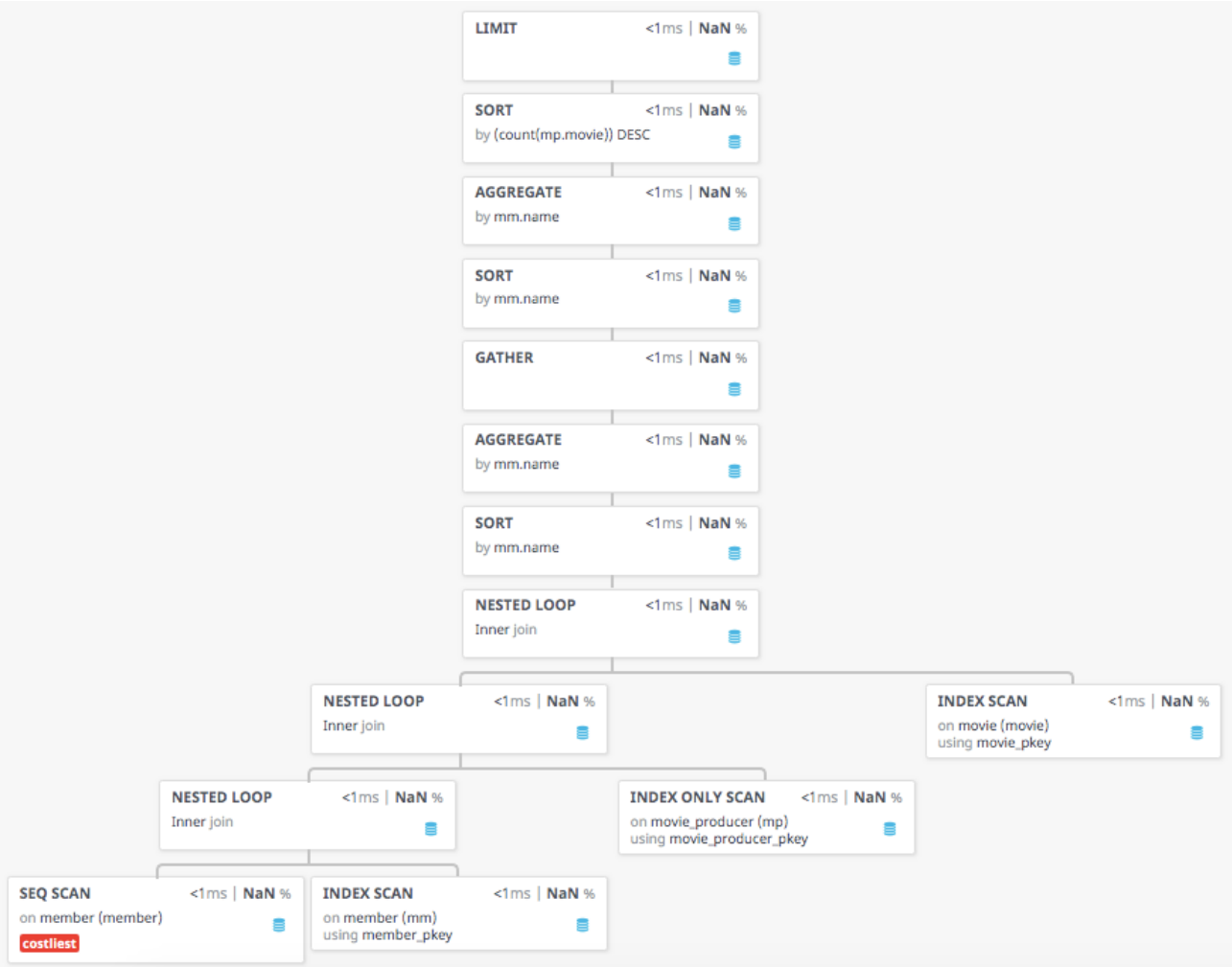
2.3.

```
select mm.name, count(mp.movie) as ct from movie_producer as
mp, member as mm
where
exists(select * from movie where startYear = 2017 and movie.id =
mp.movie)
and
exists(select * from member where name like '%Gill%' and
member.id = mp.producer
and mp.producer = mm.id)
group by mm.name
order by ct desc
limit 1;
```

output of program:

*Finish retrieve producers who have produced the most talk shows in 2017 and whose name contains “Gill” in 2014*
*in **12.78** seconds*
*result:*
*name: Ryan Gill;number of talk show produced: 81*

As for this query, it use two subquery. In one it finds all producers who produce movies of 2017. In another one, it finds producers whose name contains “Gill”. With these two constrains, we use count function with group by producer to get the producer and amount of his production. Then it do sorting and get the first one, which would be the maximum. As is shown in the picture, it does join cause we have join in subquery. And several sort on count. While we do group by, it sort the name and sum the “value” of each name together.




2.4

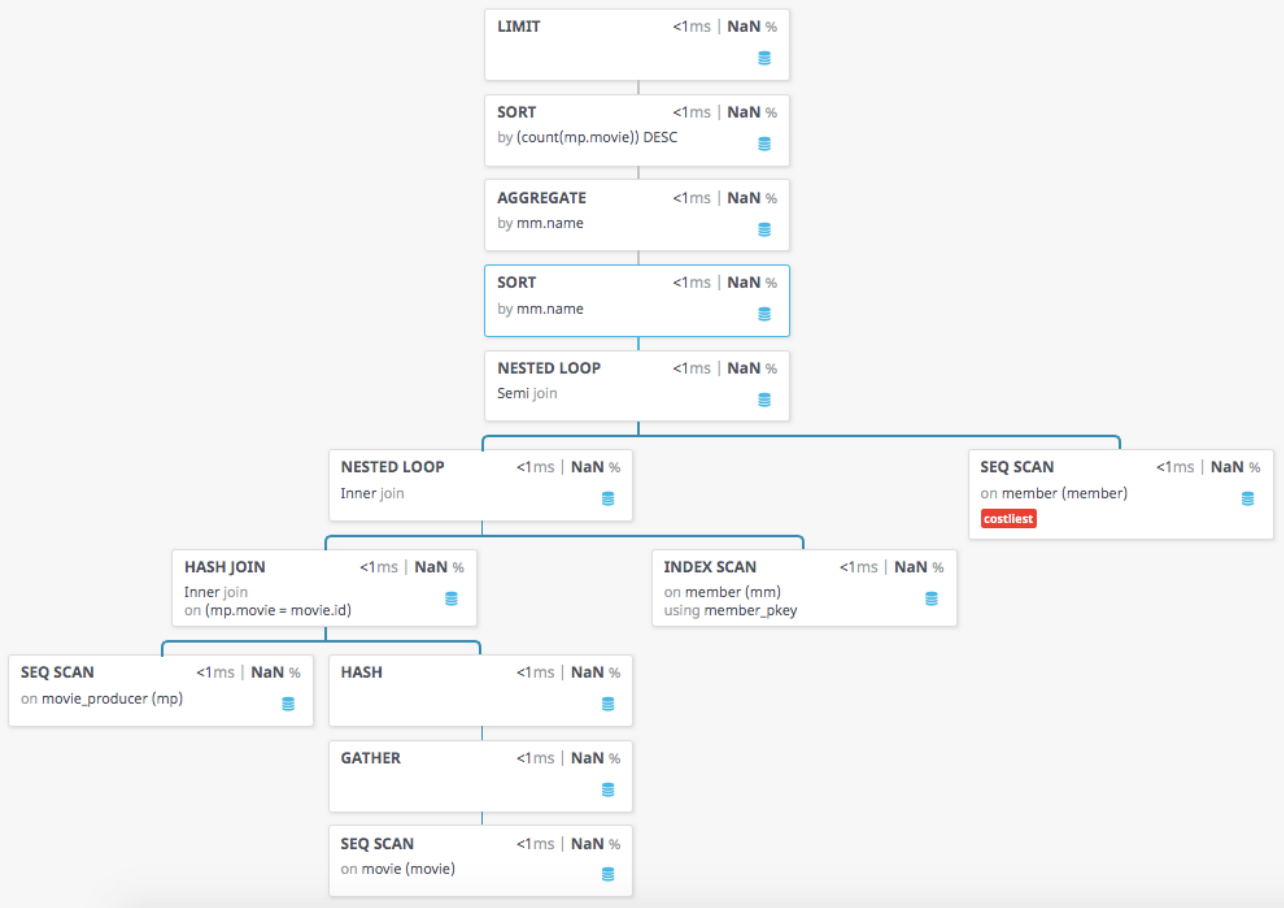
```
select mm.name, count(movie) as ct from movie_producer as mp,
member as mm
where exists (select * from movie where runtime > 120 and movie.id
= mp.movie)
and exists (select * from member where deathYear is null and
mm.id = mp.producer)
and mp.producer = mm.id
group by mm.name
order by ct desc
limit 1;
```

output of program:

*Finish retrieve alive producers with the greatest number of long-run movies produced*
*in **36.59** seconds*

*result:*
*name: Vince McMahon;number of movie produced: 140*

The idea of this query is kind of like query 3. I do use exists statement to cast constrains on our query. The first subquery gets all movie running longer than 120 minutes and being produced by those producers. The second subquery gets alive producers. Then we join table movie_producer with member to query producer's name. By sorting it, we get the maximum number of movies the producer produced. In the picture, our costliest part is scanning member table. As for other part, since exists statement doesn't care about what is the result but if the result exists, we improve the performance of query.

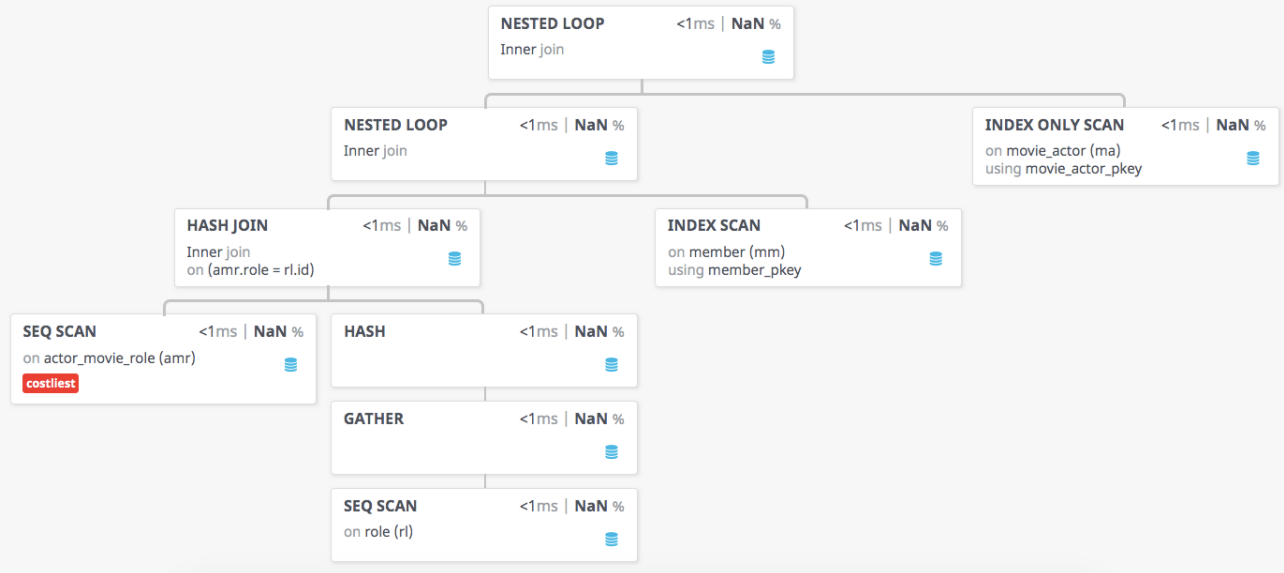



2.5
select distinct mm.name, rl.role
from actor_movie_role as amr join member as mm
on mm.id = amr.actor

```
join role as rl
on rl.id = amr.role
where (rl.role like '%Jesus%' or rl.role like '%Christ%') and
mm.deathYear is null;
```

output of program:
*Finish retrieve alive actors who have portrayed Jesus Christ (look for both words independently)*
*in **24.14** seconds*
*result:*
*number of results: 6029*
*Few examples are shown below.*
*George Cheung, Christopher*
*Sam Barriscale, Christopher Laslett*
*Owen Coomer, Christopher*
*Angelo Jurkas, Jesus*
*Chris Shields, Roman Christophe*

In this query, we do join on three tables. The join between member and actor_movie_role is to get the name of actor. Then the join between role and actor_movie_role is to query the actor who played the role. In where clause, we get alive actors and check if they played roles whose name contains “Jesus Christ”. In the picture, it shows the costliest part is scaring actor_movie_role.



5.

for query 1:

Since in both table movie_actor and actor_movie_role, all attributes of them are primary key. So the database has already created index on them.

for query 2:

create index index_name on member(name)

The finish time in question 2 of query 2 is 15.45 seconds.

After creating index on name, the time has reduced to about 9 seconds. In this query we do distinct query of name and this is improved by index.

```
Finish retrieve alive actors whose name starts with "Phi" and did not participate in any movie in 2014
 in 8.95 seconds
number of results: 4165
Few examples are shown below.
Phi Bulani
Phi Clarke
Phi Hung Nguyen
Phi Huynh
Phi Lan
```

for query 3:

Since we have already create index on member(name) and we don't use other attributes that's not primary key in this query, we don't create additional index. We can also see that after creating index on member(name), the time has reduced to 5.43 seconds, while in the previous result it is 12.78 seconds. This really great.

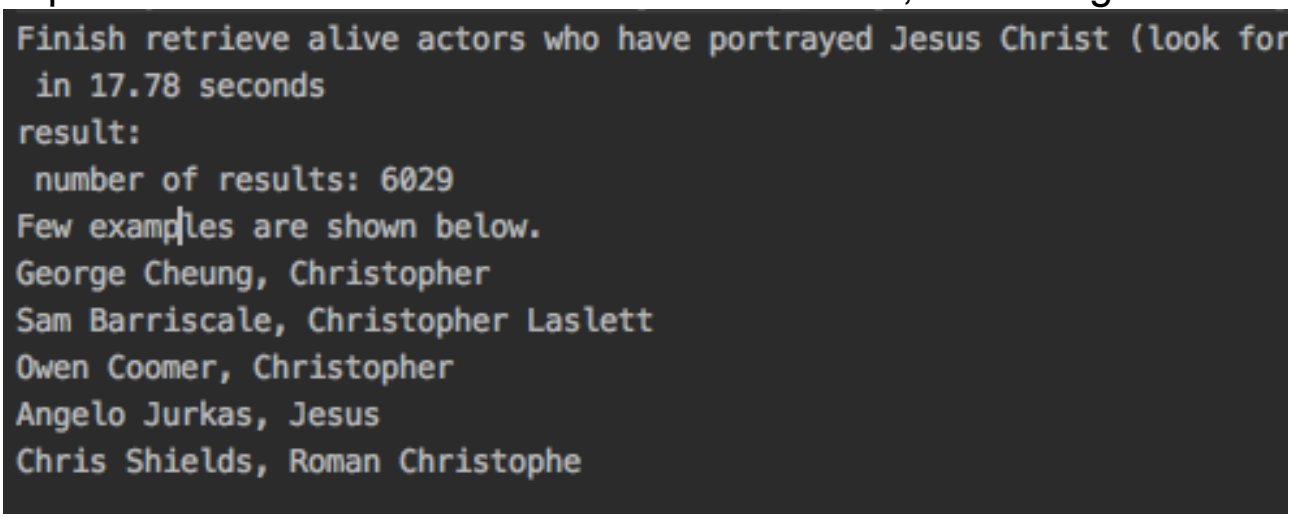
```
Finish retrieve producers who have produced the most talk shows in 2017 and whose name contains "Gill" in 2014
 in 5.43 seconds
result:
 name: Ryan Gill;number of talk show produced: 81
```

for query 4:

We don't add additional index on table. Same reason as query 3. We can also see that result of 12.99 seconds is better than previous result of 36.59 seconds.

```
Finish retrieve alive producers with the greatest number of long-run movies produced
in 12.99 seconds
result:
 name: Vince McMahon;number of movie produced: 140
```

for query 5:
I add an index on role(role):
create index on role(role)
Because this statement query role, the index will improve the query performance. We can see after creating index the time has improved from 24.14 seconds to 17.78 seconds, which is good too.




By looking up some materials, I see the index can improve query since the database server will create a balance tree, whose nodes is the index value. We know that search through a balance-tree, O(logn), is better than linear search,O(n).