Name: Xinyue Liu

Student ID: 1332-0443-43

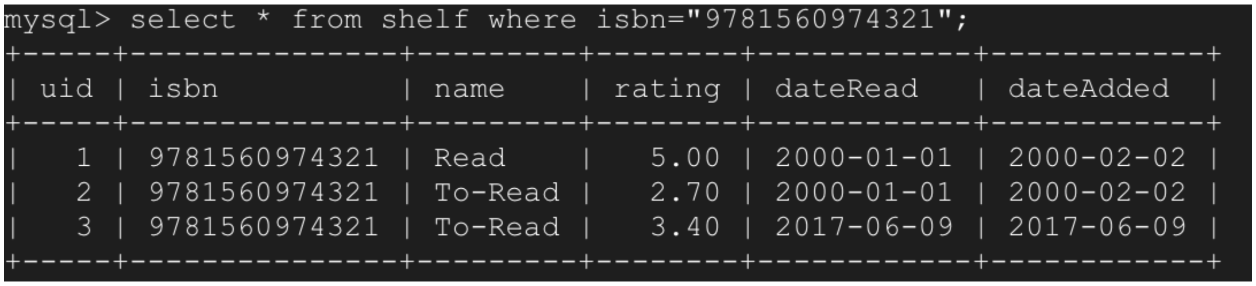
DB used: MySQL

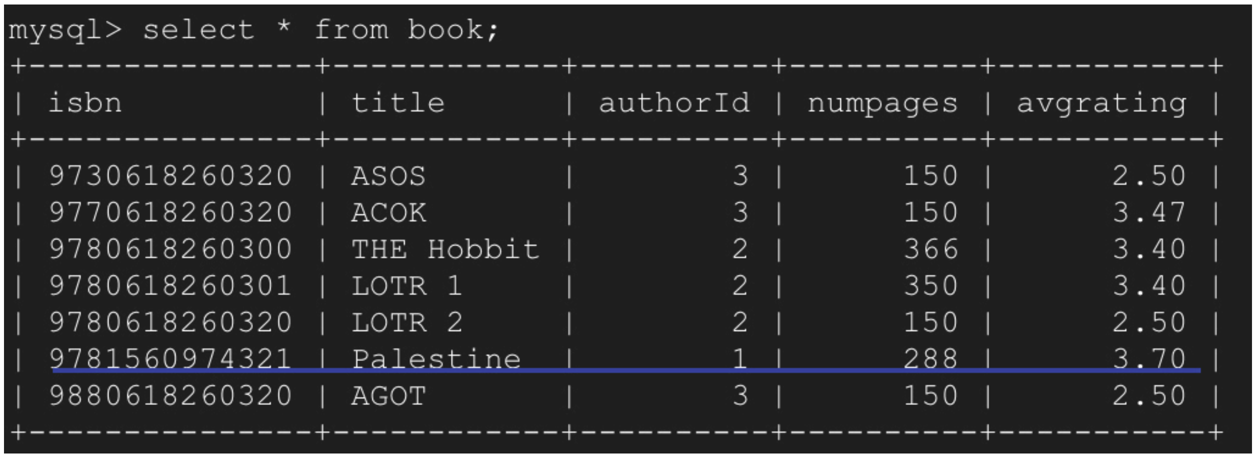
Consider Set for bonus: Set 3

Set 1-1: User adds a new book to his shelf with a rating. Update the average rating of that book.

1. Assume user 3 add a new book to his shelf
   the insert query:
```
insert into shelf(uid,isbn,name,rating,dateRead,dateAdded)
values(3,9781560974321,"To-Read",3.40,"2017-06-09","2017-06-09");
```
   update user 3's info:
```
UPDATE users set toReadCt=toReadCt+1 where uid=3;
```
2. Update the avgrating of that book:
```
Update book
set avgrating=(select SUM(rating)/COUNT(rating) from shelf where
isbn="9781560974321")
where isbn="9781560974321";
```

Explanation:
The inner query is used for counting the new average rating of this target book, and the outer query is used for updating the corresponding value we get from the inner query.

Result:
Insertion:

```
mysql> select * from shelf where isbn="9781560974321";
+-----+---------------+---------+--------+------------+------------+
| uid | isbn          | name    | rating | dateRead   | dateAdded  |
+-----+---------------+---------+--------+------------+------------+
|   1 | 9781560974321 | Read    |   5.00 | 2000-01-01 | 2000-02-02 |
|   2 | 9781560974321 | To-Read |   2.70 | 2000-01-01 | 2000-02-02 |
|   3 | 9781560974321 | To-Read |   3.40 | 2017-06-09 | 2017-06-09 |
+-----+---------------+---------+--------+------------+------------+
```
Update:



```
mysql> select * from book;
+---------------+------------+----------+----------+-----------+
| isbn          | title      | authorId | numpages | avgrating |
+---------------+------------+----------+----------+-----------+
| 9730618260320 | ASOS       |        3 |      150 |      2.50 |
| 9770618260320 | ACOK       |        3 |      150 |      3.47 |
| 9780618260300 | THE Hobbit |        2 |      366 |      3.40 |
| 9780618260301 | LOTR 1     |        2 |      350 |      3.40 |
| 9780618260320 | LOTR 2     |        2 |      150 |      2.50 |
| 9781560974321 | Palestine  |        1 |      288 |      3.70 |
| 9880618260320 | AGOT       |        3 |      150 |      2.50 |
+---------------+------------+----------+----------+-----------+
```

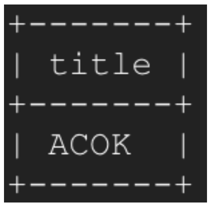
Set 1-2: Find the names of the common books that were read by any two users X and Y.

1. Query:

```
select DISTINCT title
from book,
(select shelf1.isbn
from shelf shelf1, shelf shelf2
where shelf1.isbn=shelf2.isbn
AND shelf1.uid!=shelf2.uid
AND shelf1.name=”Read”
AND shelf2.name=”Read”) result
where book.isbn=result.isbn;
```

Explanation:
From the inside out, self join shelf table to find isbn of books read by any two users, finally join with book to get distinct titles of books with same isbn.

Result:



```
+-------+
| title |
+-------+
| ACOK  |
+-------+
```

Set 1-3: For all books that user 2 has in his to-read list, find the number of friends that have already read each book and list them by book names.

(*For each book that user X has* in *his to-read list, find the number of friends that have read the book.*)

1. Query:

```
select DISTINCT title, count(title) number
from (select result1.isbn from (select isbn from shelf where uid=2 AND
name="To-Read") result1,(select isbn from shelf,(select fid from
friends where uid=2) result where shelf.uid=fid AND name="Read")
result2 where result1.isbn = result2.isbn) result3,book
```

```
where result3.isbn=book.isbn
group by title;
```

Explanation:
From inside out, result includes all ids from User 2's friends. Result1 includes all books' isbn that user 2 has in his to-read list. Result2 is the join between result and shelf, which includes all books' isbn that user 2's friends has in their read list. Result3 is the join between result1 and result2, which includes all the isbn of books that User2 doesn't read but his friends does. Finally answer is the join between result3 and book to get the corresponding book titles and also count the number by GROUP BY.

Result:
```
+-----------+--------+
| title     | number |
+-----------+--------+
| ACOK      |      1 |
| Palestine |      1 |
+-----------+--------+
```

Set 1-4: Find the names of all books where user 1 and any of his/her friends X gave the book a rating of 5 stars. (e.g., if X and Y gave 5 stars to book1, and X and Z gave 5 stars to book2, both books 1 and 2 should be included in the output.)

1. Query:
```
select DISTINCT title from book,(select result1.isbn from (select *
from shelf where uid=1)result1,(select * from shelf,(select fid from
friends where uid=1)result where shelf.uid=result.fid) result2 where
result1.isbn=result2.isbn AND result1.rating=5.00 AND
result2.rating=5.00) result3 where book.isbn=result3.isbn ;
```

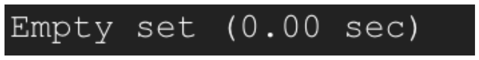
Explanation:
From inside out, result includes all the friends' id of user1. Result1 includes shelf records from user1. Result2 includes the shelf records from user 1's friends, which is the join result between result and shelf. Result3 is the final answer.

Result:
```
+-------+
| title |
+-------+
| ACOK  |
+-------+
```

Set 1-5: Find users who have only read books that have more than 300 pages.

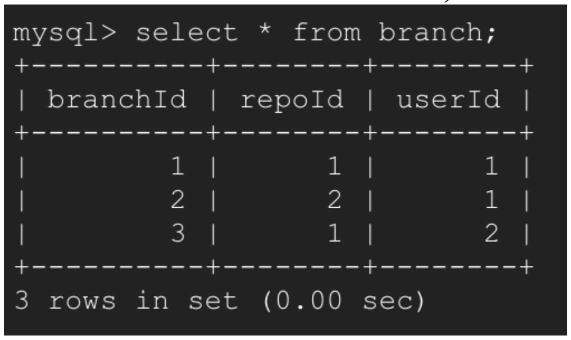
1. Query:
```
select name from users,(select DISTINCT uid from shelf,(select isbn
from book where numpages>300) result1 where shelf.isbn=result1.isbn AND
name="Read" AND uid not in (select DISTINCT uid from shelf,(select isbn
from book where numpages<=300) result2 where shelf.isbn=result2.isbn
AND name="Read")) result3 where users.uid=result3.uid;
```

Explanation:
From inside out, result1 includes all the uid of users who read books over 300 pages. Reuslt2 includes all the uid of users read books with no more than 300 pages. The users uid must be in result1 and must not be in result2. And join with table users to get the corresponding user name.
Result:



```
Empty set (0.00 sec)
```

Set 2: the table branch I created, which is not included in the sample tables.



```
mysql> select * from branch;
+----------+--------+--------+
| branchId | repoId | userId |
+----------+--------+--------+
|        1 |      1 |      1 |
|        2 |      2 |      1 |
|        3 |      1 |      2 |
+----------+--------+--------+
3 rows in set (0.00 sec)
```

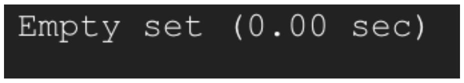
Set 2-1: Find the commits made by user 1 in all branches of a repository between 1st May and 10th May.

1. Query:

```
select commitId,commits.branchId,commitTime,noOfFiles,additions,deletions
from commits,(select branchId from branch where repoId=1 AND
userId=1)result where commits.branchId=result.branchId AND
commits.commitTime >= '2000-05-01' AND commits.commitTime <='2000-05-10';
```

Explanation:
From inside out, result includes all the branch id of User 1's repository 1 (repoId=1). Then join result and commits to select the qualified commits.

Result:



```
Empty set (0.00 sec)
```

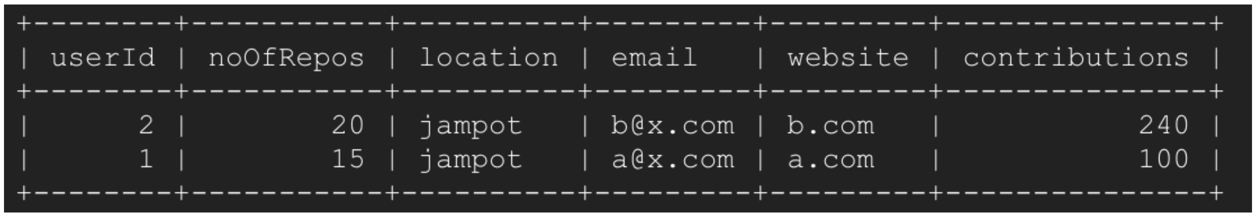
Set 2-2: Display the top ten users who made the most contributions

1. Query:

```
select * from (select * from users order by contributions desc) result
limit 10;
```

Explanation:
From inside out, result is the table users reordered by contributions, and then use limit to select top 10 of the result.

Result:



```
+--------+-----------+----------+---------+---------+---------------+
| userId | noOfRepos | location | email   | website | contributions |
+--------+-----------+----------+---------+---------+---------------+
|      2 |        20 | jampot   | b@x.com | b.com   |           240 |
|      1 |        15 | jampot   | a@x.com | a.com   |           100 |
+--------+-----------+----------+---------+---------+---------------+
```

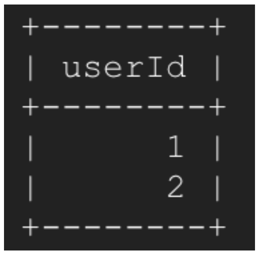
Set 2-3: Find the users who made branches of either of repositories 1 (repoId=1) or 2 (repoId=2) but not of a repository 3 (repoId=3).

1. Query:

```
select distinct userId from (select userId from branch where repoId=1
OR repoId=2) result1 where userId Not in (select userId from branch
where repoId=3);
```

Explanation:
Result1 is the userId of users who made branches of either of repositories1 or 2. Selecting those users which are not in the list of users who made branches of repository3 is the answer.

Result:



```
+--------+
| userId |
+--------+
|      1 |
|      2 |
+--------+
```

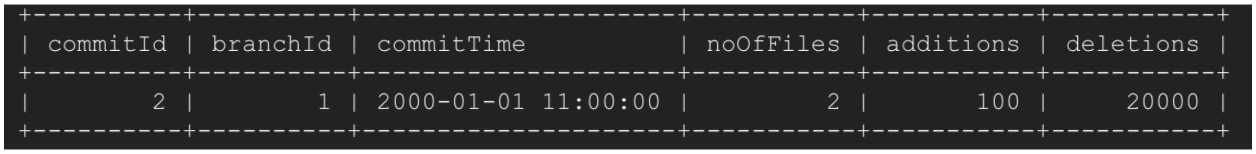
Set 2-4: Find the top commit with the highest lines of code reduced. (Hint: We need to find the maximized value of: number of deletions - number of additions in each commit).

1. Query:

```
select commitId,branchId,commitTime,noOfFiles,additions,deletions from
(select *,deletions-additions reduceNum from commits order by reduceNum
DESC) result limit 1;
```

Explanation:
Result is the table commits plus one new column, which shows the value of number of deletions - number of additions. And this result table is in descending order by this new column's values. Therefore, the answer will be the first row of all the columns from result except reduceNum.

Result:



```
+----------+----------+---------------------+-----------+-----------+-----------+
| commitId | branchId | commitTime          | noOfFiles | additions | deletions |
+----------+----------+---------------------+-----------+-----------+-----------+
|        2 |        1 | 2000-01-01 11:00:00 |         2 |       100 |     20000 |
+----------+----------+---------------------+-----------+-----------+-----------+
```

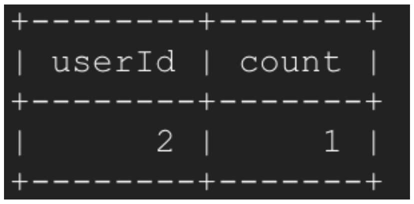
Set 2-5: List the users who solved more issues than they raised. (i.e. number of issues in which they were the resolver is greater than the number of issues where they were the creator.)

1. Query:

```
select resolverId userId, ctResolve-ctCreate count from(select resolverId,
count(resolverId) ctResolve from issue group by(resolverId))resolver,(select
creatorId,count(creatorId)ctCreate from issue group by(creatorId))creator
where resolverId=creatorId AND ctResolve-ctCreate>0 order by count DESC;
```

Explanation:
Resolver table contains the resolvers' Id and the number of case(s) resolved by him or her. Creator table contains the creators' Id and the number of case(s) created by him or her. Then join those two table to figure out the positive differences (resolve case num – create case num) of each user.

Result:



```
+--------+-------+
| userId | count |
+--------+-------+
|      2 |     1 |
+--------+-------+
```

Set 3:
DDL:

```
create database set3;
use set3;

create table luser(userId int, email varchar(30) not null,handle
varchar(50),name varchar(20),headline varchar(50),location
varchar(50),primary key(userId),unique(email));
create table network(userId int not null, friendId int not null,dateAdded
date, note varchar(100),primary key(userId,friendId));
create table application(userId int, jobId int, primary key(userId,jobId));
create table organization(organizationId int, name varchar(50),primary
key(organizationId));
create table experience(experienceId int, userId int,title
varchar(30),organizationId int, fromD date, toD date,location
varchar(50),description varchar(100),primary key(experienceId));
create table job(jobId int not null, datePosted date, organizationId int,
genre varchar(50), primary key(jobId));
alter table network add foreign key(userId) references luser(userId);
alter table application add foreign key(userId) references luser(userId);
alter table application add foreign key(jobId) references job(jobId);
alter table experience add foreign key(userId) references luser(userId);
```

```
create table yuser(userId int not null, email varchar(30) unique, phone
varchar(10),handle varchar(50),name varchar(30),favoriteGenre
varchar(50),primary key(userId));
create table subscription(channelId int, userId int, primary key(channelId,
userId));
create table channel(channelId int, genre varchar(50), ownerId int,
videoCount int, subscriberCount int, primary key(channelId));
create table video(videoId int, channelId int, primary key(videoId));
alter table subscription add foreign key (userId) references yuser(userId);
alter table subscription add foreign key (channelId) references
channel(channelId);
alter table video add foreign key (channelId) references channel(channelId);
```

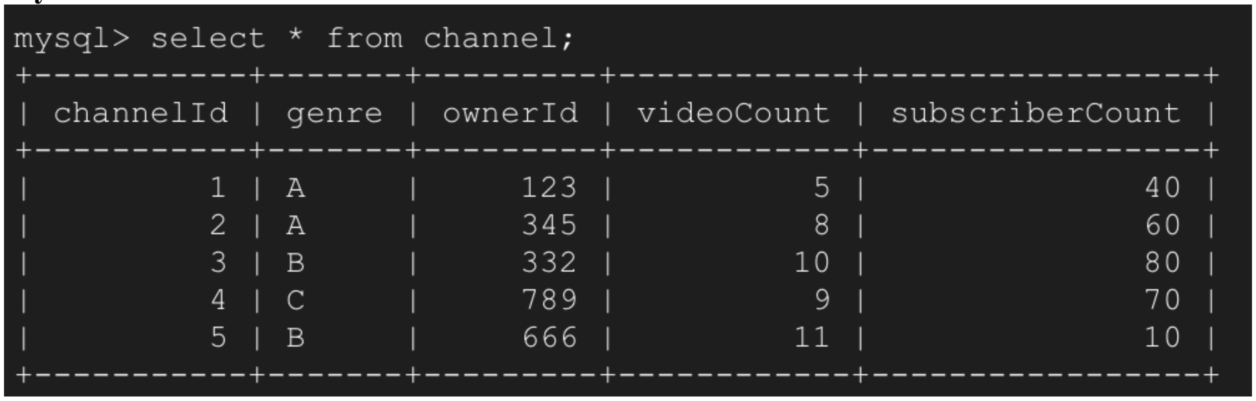
Set 3-1: List all channels that have more than 50 videos and have the same genres as jobs searched on LinkedIn by user X (userId=1).

Assume the videoCount attribute and subscriberCount get updated properly once the database has changed.

1. Query:
   ```
   select channelId from channel,(select genre from job,(select jobId from
   application where userId=1) result where job.jobId=result.jobId) result2
   where channel.genre=result2.genre AND videoCount>50;
   ```

   Explanation:
   The result is the all the job id of jobs user X has recorded. Result2 is the result1's corresponding genres table. Join result2 with channel to select the qualified channel by using where.

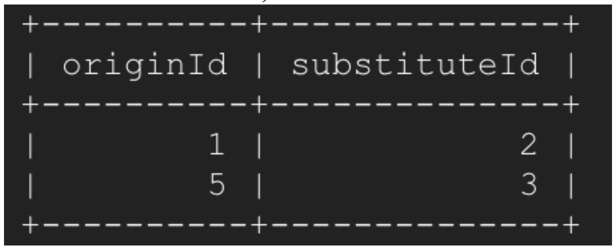
Set 3-2: Delete all channels that have less than 50 subscribers but move their videos to the channel with highest number of subscribers in the same genre.
My channel table:



```
mysql> select * from channel;
+-----------+-------+---------+------------+-----------------+
| channelId | genre | ownerId | videoCount | subscriberCount |
+-----------+-------+---------+------------+-----------------+
|         1 | A     |     123 |          5 |              40 |
|         2 | A     |     345 |          8 |              60 |
|         3 | B     |     332 |         10 |              80 |
|         4 | C     |     789 |          9 |              70 |
|         5 | B     |     666 |         11 |              10 |
+-----------+-------+---------+------------+-----------------+
```

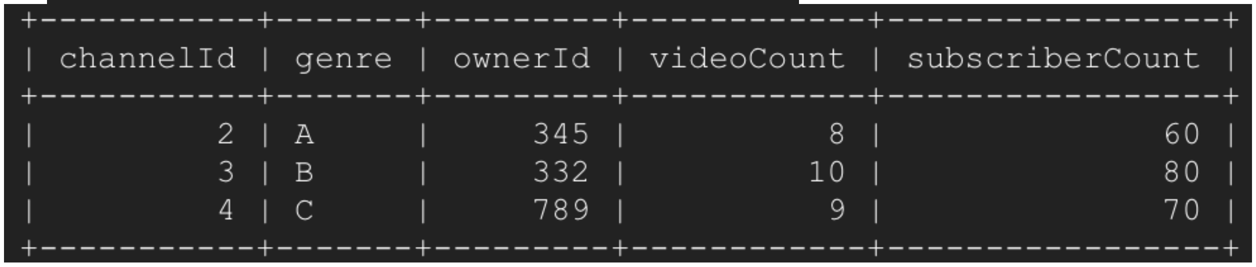
1. Query:
   ```
   select result1.channelId originId,result2.channelId substituteId from
   (select channelId,genre from channel where subscriberCount
   <50)result1,(select channelId,channel.genre from channel,(select
   genre,MAX(subscriberCount)max from channel group by genre)result where
   channel.genre=result.genre AND
   channel.subscriberCount=result.max)result2 where
   result1.genre=result2.genre;
   ```
```
This above query I can get a table which includes the channels' id which
should be deleted and their corresponding substitute channelId for video
table.
```

Explanation:
The result contains all genre and their corresponding Max subscriberCount. Result1 contains all channels' id that have less than 50 subscribers and their corresponding genre. Result2 is the join of channel and result2, which contains the channels' id with highest number of subscribers group by genre. And the query will get a table of two columns. The first one indicates those channelIds need to be deleted, and the second one shows their corresponding substitute channelId, like below:



```
+----------+--------------+
| originId | substituteId |
+----------+--------------+
|        1 |            2 |
|        5 |            3 |
+----------+--------------+
```

Then we can based on this table, update video table one by one. For example,
```
update video
set channelId=2
where channelId=1;
```
Finally, we can delete those channels that have less than 50 subscribers
```
delete from channel where subscriberCount < 50;
```



```
+-----------+-------+---------+------------+-----------------+
| channelId | genre | ownerId | videoCount | subscriberCount |
+-----------+-------+---------+------------+-----------------+
|         2 | A     |     345 |          8 |              60 |
|         3 | B     |     332 |         10 |              80 |
|         4 | C     |     789 |          9 |              70 |
+-----------+-------+---------+------------+-----------------+
```

Set 3-3: Find all jobs on LinkedIn whose genre matches Owner-ID X's channel's genre.

1. Query
   ```
   select jobId,datePosted,organizationId,job.genre from job,(select genre
   from channel where ownerId=789) result where job.genre=result.genre;
   ```

   Explanation:

The result includes all the genres which Owner-ID X has. Join result with table job to get those jobs on LinkedIn whose genre matches Owner-ID X's channel's genre.

Set 3-4: Find friends of friends of User X on LinkedIn who have subscribed to channels with the genre that matches user X's favorite genre on YouTube. (This can be used to suggest friends.)

```
select name from luser,(select 2LevelFriendEmail from (select
favoriteGenre from yuser where email=(select email from luser where
userId=1))result7,(select 2LevelFriendId, 2LevelFriendEmail,
channel.genre 2LevelFriendGenre from channel,(select 2LevelFriendId,
2LevelFriendEmail, channelId from subscription,(select userId
2LevelFriendId,2LevelFriendEmail from yuser,(select email
2LevelFriendEmail from luser,(select network.friendId 2LevelFriend from
network,(select friendId from network where userId=1) result where
network.userId=result.friendId) result1 where
luser.userId=result1.2LevelFriend) result3 where
yuser.email=result3.2LevelFriendEmail) result4 where
subscription.userId=result4.2LevelFriendId) result5 where
channel.channelId=result5.channelId) result6 where
result7.favoriteGenre=result6.2LevelFriendGenre) result8 where
luser.email=result8.2LevelFriendEmail;
```

Explanation:
Result->user1's friends id list
Result1->user1's friends' friends' id list
Reuslt3-> user1's friends' friends' email list
Result4-> user1's friends' friends' userId (yuser table) & email list
Result5-> user1's friends' friends' userId (yuser table) & email & channelId (subscribed channel) list
Result6-> user1's friends' friends' userId (yuser table) & email & genre list
Result7->user1's favorite genre
Result8->join result7 and result6 to get the list of email of users who have subscribed to channels with the genre that matches user X's favorite genre on YouTube
Answer->join result8 and luser to get the corresponding name of those emails we got from result8.

Set 3-5: Add a column "location" to YouTube's User table based on the location associated with their LinkedIn account.

```
alter table yuser add location varchar(50);
select yuser.userId, luser.location from luser,yuser where
luser.email=yuser.email;

Then, update the location column of yuser based on the above result.
For instance,

update yuser
set locatin="LA,CA"
where userId=1
```

Explanation:

First, add a column “location” to YouTube’s User table. Then get all the location information and their corresponding userId of YouTube’s User table. Then, update the location column of yuser based on the above result one by one.