CSCI 620

ASSIGNMENT 3

Abhishek Shah, as5553

Question 1

```
-- Create new relation for Question 1
create table Question1(
        Titleid INT,
        type varchar(255),
        startYear INT,
        runtime INT,
        avgRating FLOAT,
        Genreid INT,
        genre varchar(255),
        Memberid INT,
        birthYear SMALLINT,
        character INT
);


-- Create Temp table
create table Temp(title integer);

-- Insert into Temp table
insert into Temp
select title
from actor_title_character
group by title,actor
having count (character) > 1;

-- Insert into main table acc. to given conditions
insert into Question1
select t.id, t.type, t.startYear, t.runtime, t.avgRating, g.id, g.genre, m.id, m.birthYear, atc.character
from Title as t
join Title_Genre as tg
on t.id = tg.title
join Genre as g
on tg.genre = g.id
join Title_Actor as ta
on ta.title = t.id
join Member as m
```

```
on m.id = ta.actor
join Actor_Title_Character as atc
on atc.actor = m.id and atc.title = t.id
left outer join tmc on tmc.title=t.id
where t.runtime >= 90 and t.type='movie' and tmc.title is null;

-- add a new column id and make it the primary key
alter table Question1 add primary key(Titleid, Memberid, Genreid);
```

Description

A new relation is created by joining the tables Title, Title_Actor, Genre, Member, Actor_Title_Character on their primary keys.

After joining the table, a **where** condition is used to filter the **runtime** minutes of the movie to be greater than or equal to 90 minutes.
Another condition was to select only the actors with single character.

The subquery written handles actors with only single characters. They are taken and are then grouped by the movies and actors of the Actor_Title_Character table so that each and every actor who has worked for a particular Title(id) is taken.

The inner subquery takes care of the single character clause. A temp table is created where the actors with characters greater than one are taken and stored.

The table is left joined when you map all the values of the titles in the **temp** table to the values in the **Title** tables and whichever values that correspond to null are eventually considered as the actors that played single character.

Question 2

```
import itertools
import psycopg2
def connection(host, database, user, password, port):
    try:
        connection = psycopg2.connect(
            host=host,
            database=database,
            user=user,
            password=password,
            port=port)
        if connection:
            print("Connection Successful!")
            return connection
    except Exception as err:
        print(err)
def columnNames(connection):
    cursor = connection.cursor()
    select_query = """
    SELECT column_name
    FROM information_schema.columns
    WHERE table_schema = 'public'
    AND table_name    = 'question1'
    """
    cursor.execute(select_query)
    columns = []
    for column in cursor:
        columns.append(column[0])
        yield column[0]
def combinations(col):
    combi = []
    for x in range(1, len(col) + 1):
        for ss in itertools.combinations(col, x):
            combi.append(ss)
    finalList = [','.join(i) for i in combi]
    return finalList
def funcDependencies(connection, table_name):
    nameColumn = list(columnNames(connection))
    combiList = combinations(nameColumn)
    print(combiList)
    cursor = connection.cursor()
    func_depends = []
    for i in range(0, len(nameColumn)):
        for j in range(0, len(combiList)):
            if nameColumn[i] in combiList[j]:
                continue
            fd_query = f""" SELECT {combiList[j]}
            FROM question1
            GROUP BY {combiList[j]}
            HAVING COUNT(DISTINCT {nameColumn[i]}) > 1
            """
            cursor.execute(fd_query)
            if cursor.rowcount == 0:
                func_depends.append(f'{combiList[j]} -> {nameColumn[i]}')
                print(f'{combiList[j]} -> {nameColumn[i]}')
    if func_depends:
        print('Functional dependencies are:')
        for fd in func_depends:
            print(fd)
    else:
        print('Not found!')
if __name__ == "__main__":
    db_connection = connection('localhost', 'imdb6202',
                               'postgres', password='Abhishek@123', port=5432)
    tables_to_examine = list(columnNames(db_connection))
    for table in tables_to_examine:
        funcDependencies(db_connection, table)
```

Description

The naive approach is to check all the pair of rows that is each tuple in each column.

The algorithm used in this naive approach uses each column combination in X-> Y and for each pair of tuples in the column (t1,t2) we check the condition that, t1[X] = t2[Y] and t1[Y] != t2[Y].

This algorithm is to fetch all the functional dependencies using the naive approach.

Here the condition given was to fetch all functional dependencies with only one attribute on the right hand side.

After establishing the connection to the PostgreSQL server, itertool combinations is used to find all possible combinations of rows and columns.

Using information schema of the new relation created for question 1, the column names are retrieved and stored as columns. Using the itertool combinations all the possible combinations are found and stored in a list.

Here we also needed to take care of the trivial dependencies which we check by checking that if value in nameColumn is also in CombiList then we just continue.

That is, all trivial functional dependencies like,

A, B $\rightarrow$ A

are ignored.

For fetching the functional dependencies, we then iterate with each column in the headers where each x row is iterated till the length of the columns and each y row is iterated with the length of the combinations is taken.

Functional dependency is found if it's not the case that there's more than one value in nameColumn associated with CombiList that is, if the count is 0 then the two rows are functionally dependent.

Estimated time

With respect to time complexity,

Time taken to run a single column with all possible combinations: 2 hours

Total time taken for 10 columns with all combinations: 20 hours

Question 3

-- add a new column id and make it the primary key
alter table Question1 add primary key(Titleid, Memberid, Genreid);

```
import itertools
import psycopg2
def connection(host, database, user, password, port):
    try:
        connection = psycopg2.connect(
            host=host,
            database=database,
            user=user,
            password=password,
            port=port)
        if connection:
            print("Connection Successful!")
            return connection
    except Exception as err:
        print(err)
def columnNames(connection):
    cursor = connection.cursor()
    select_query = """
    SELECT column_name
    FROM information_schema.columns
    WHERE table_schema = 'public'
    AND table_name    = 'question1'
    """
    cursor.execute(select_query)
    columns = []
    for column in cursor:
        columns.append(column[0])
        yield column[0]
def combinations(col):
    combi = []
    for ss in itertools.combinations(col, 1):
        combi.append(ss)
    for x in itertools.combinations(col, 2):
        combi.append(x)
    finalList = [','.join(i) for i in combi]
    return finalList
def funcDependencies(connection, table_name):
    nameColumn = list(columnNames(connection))
    combiList = combinations(nameColumn)
    print(combiList)
    left_side = []
    right_side = []
    l = len(combiList)
    print(l)
    cursor = connection.cursor()
    func_depends = []
    for i in range(0, len(nameColumn)):
        for j in range(0, len(combiList)):
            if nameColumn[i] in combiList[j]:
                continue
            flag = False
            for x in range(len(left_side)):
                if left_side[x] in combiList[j] and nameColumn[i] == right_side[x]:
                    flag = True
                    break
            if flag:
                left_side.append(combiList[j])
                right_side.append(nameColumn[i])
                print(combiList[j] + '->' + nameColumn[i])
                continue
            fd_query = f""" SELECT {combiList[j]}
                            FROM question1
                            GROUP BY {combiList[j]}
                            HAVING COUNT(DISTINCT({nameColumn[i]})) > 1
                            """
            cursor.execute(fd_query)
            if cursor.rowcount == 0:
                left_side.append(combiList[j])
                right_side.append(nameColumn[i])
                print(combiList[j] + '->' + nameColumn[i])
if __name__ == "__main__":
    db_connection = connection('localhost', 'imdb6202',
                               'postgres', password='Abhishek@123', port=5432)
    tables_to_examine = list(columnNames(db_connection))
    for table in tables_to_examine:
        funcDependencies(db_connection, table)
```

Output

```
titleid->type
startyear->type
runtime->type
avgrating->type
genreid->type
genre->type
memberid->type
birthyear->type
character->type
titleid,startyear->type
titleid,runtime->type
titleid,avgrating->type
titleid,genreid->type
titleid,genre->type
titleid,memberid->type
titleid,birthyear->type
titleid,character->type
startyear,runtime->type
```

startyear,avgrating->type
startyear,genreid->type
startyear,genre->type
startyear,memberid->type
startyear,birthyear->type
startyear,character->type
runtime,avgrating->type
runtime,genreid->type
runtime,genre->type
runtime,memberid->type
runtime,birthyear->type
runtime,character->type
avgrating,genreid->type
avgrating,genre->type
avgrating,memberid->type
avgrating,birthyear->type
avgrating,character->type
genreid,genre->type
genreid,memberid->type
genreid,birthyear->type
genreid,character->type
genre,memberid->type
genre,birthyear->type
genre,character->type
memberid,birthyear->type
memberid,character->type
birthyear,character->type
titleid->startyear
titleid,type->startyear
titleid,runtime->startyear
titleid,avgrating->startyear
titleid,genreid->startyear
titleid,genre->startyear
titleid,memberid->startyear
titleid,birthyear->startyear
titleid,character->startyear
titleid->runtime
titleid,type->runtime
titleid,startyear->runtime
titleid,avgrating->runtime
titleid,genreid->runtime
titleid,genre->runtime
titleid,memberid->runtime
titleid,birthyear->runtime
titleid,character->runtime
titleid->avgrating
titleid,type->avgrating
titleid,startyear->avgrating
titleid,runtime->avgrating

titleid,genreid->avgrating
titleid,genre->avgrating
titleid,memberid->avgrating
titleid,birthyear->avgrating
titleid,character->avgrating
genre->genreid
titleid,genre->genreid
type,genre->genreid
startyear,genre->genreid
runtime,genre->genreid
avgrating,genre->genreid
genre,memberid->genreid
genre,birthyear->genreid
genre,character->genreid
memberid->birthyear
titleid,memberid->birthyear
type,memberid->birthyear
startyear,memberid->birthyear
runtime,memberid->birthyear
avgrating,memberid->birthyear
genreid,memberid->birthyear
genre,memberid->birthyear
memberid,character->birthyear
titleid,memberid->character

Description

For the following functional dependency

titleid -> runtime

is a functional dependency,
then with the use of lattice, we are gonna ignore all the functional dependencies of the type,

titleid,type->runtime
titleid,startyear->runtime
titleid,avgrating->runtime
titleid,genreid->runtime
titleid,genre->runtime
titleid,memberid->runtime
titleid,birthyear->runtime
titleid,character→runtime

By ignoring I mean, the lattice just makes it skips/ignores the functional dependency because

titleid -> runtime

was checked by query.

So, all the remaining similar functional dependencies,

titleid, type->runtime
titleid, startyear→runtime etc.

are not checked by the query and are just skipped because of the lattice.

Question 4

In the first question, we were considering all the actors with single character.
So, there was no multi valued attributes in the column character.

So for every actor-title unique combination we have a unique character(id) which will hold the valid functional dependency.

Functional dependency will be valid when any unique attribute there is also the unique attribute dependent on it.

That is, X → Y means, for every value of x there is a unique value of Y.

Now, If we assume that the characters are not to be considered single that is they have played more than one characters, then the characters column would be a multi valued attribute where for each title-actor (title, actor) combination there would be more than one character id associated with the unique combination that doesn’t hold the basic functional dependency rule.

That is, for every X -> Y, functional dependency won’t hold if we consider the actors who played more than one character.

Question 5

Candidate Key

Titleid, Genreid, Memberid

Canonical Cover

Titleid → avgrating
Titleid → avgrating
Titleid → avgrating
Titleid → avgrating
Genreid → genre
genre → Genreid
Titleid, Memberid → characters

3NF Decomposition

R1: (Titleid, Genreid, Memberid)

R2: (Genreid, Genre)

R3: (Memberid, birthYear)

R4: (Titleid, type, runtime, avgrating, startYear)

R5: (Memberid, Titleid, character)