CISS430: Database Systems
Assignment 3

OBJECTIVES
1. Design and build a moderately complex database from publicly available data sets.
2. Write complex SQL queries.
3. Write SQL queries that involve subqueries.
4. Write SQL queries that involve aggregate operators.

You are not allowed to use any material other than MySQL reference manual, my notes, and documentation on the Python language. Your are encouraged to discuss with others in the class. Discussion must be restrict to sharing ideas, not code. You must write your own code (MySQL SQL or Python).

Some python

We'll be using python 2.

[WARNING: For spring 2021, we'll be moving on to python 3.]

Here's the URL for the string module: `https://docs.python.org/2/library/string.html#module-string`

Here's a quick tutorial on working with python strings:

```
s = "hello world"
print s
t = s[:5]
print t
t = s[5:]
print t
t = s[4:10]
print t
t = s[4]
print t

print "abc  %s   def  %s   ghi %s !!!!" % ("aaaa", "bbbbb", "cccccc")

print "abc" == "abc", "abc" == "abcd"

s = "hello world !!!"
xs = s.split(' ')
print xs

print "bcd" in "abcde", "bcd" not in "abcde"

print "abcdef".find("cd")

s = "   abcdef       "
t = s.lstrip()
print "[%s]" % t
t = s.rstrip()
print "[%s]" % t
t = s.strip()
print "[%s]" % t

s = "hello world"
t = s.replace("l", "xyz")
print t
t = s.replace(" wo", ", WO")
print t
```

Here's how to read and write to files:

```
f = file("a.txt", 'w')
f.write("hello world ... 0\nhello world ... 1\n")
f.write("hello world ... 2\n")
f.close()

// you can read a.txt

g = file("a.txt", 'r')
lines = g.readlines()
print lines
for line in lines:
    print line
    print line.split(' ')
```

Here's how you work with python lists (which are arrays):

```
xs = ["abc", "def", 42, "ghijkl"]:
for i in len(xs):
    print xs[i]
for x in xs:
    print x
for i, x in enumerate(xs):
    print i, x

ys = []
for x in xs:
    ys.append(x)
    ys.append(x)
print ys

zs = ys[1:3]
print zs

xs = []
while 1:
    x = raw_input("enter a string:")
    if x == '':
        break
    xs.append(x)
    print xs
```

Here's how you use sets:

```
xs = set()
ys = []
for x in [3,2,1,4,5,2,3,1]:
    xs.add(x)
    ys.append(x)
    print xs, ys

xs.remove(3)
ys.remove(3)
print xs, ys

del xs[0]
print xs
```

From the code provided below you should be able to figure out how branching works in python.

Here's how you work with functions:

```
def f(x):
    return x + 1

def g(x):
    return [x, x]

def h(x):
    return x + "!!!"

print f(2), g(3), h("hello world")
```

Here's how you work with dictionaries (i.e., hashtables):

```
height = {}
height['john'] = 5.4
height['tom'] = 6.4
height['mary'] = 7.4

print height['mary']

height['mary'] = 7.7
print height['mary']

for key in height.keys():
    print key, height[key]

for value in height.values():
    print value

for key, value in height.items():
    print key, value

for name in ['tom', 'john', 'sue', 'mary']:
    if height.has_key(name):
        print name, height[name]
    else:
        print name, 'not in height dictionary'

del height['tom']
print height
```

Now you are ready to study the python tutorial on python website https://docs.python.org/2/tutorial/index.html. You don't really need to know all the syntax of python for this assignment. It's probably best to just read the basics in the above tutorial and go back to the tutorial only when you need a syntax.

THE IMDB DATA SETS AND YOUR DATABASE

First go to https://www.imdb.com/interfaces/. You will find some publicly available IMDB datasets. Download all the files and read the information on the webpage.

Construct a database from the IMDB datasets. Use appropriate constraints and make sure you do not have unneccessary data redundancies.

To create less confusion, for table names and column names, follow IMDB naming convention as much as possible. For instance for `name.basics.tsv`, the name of the table is `name_basics` and for `title.ratings.tsv`, the name of the table is `title_ratings`. Also, do not change the data. For instance use their movie ids instead of changing them to integers.

IMDB uses `'\N'` when a value does not exists for a field. (This should correspond to `NULL` in the database.)

Note that there are some fields which are comma-separated-values (csv). For instance in the `name.basics.tsv`, the `primaryProfession` is a csv field. In particular, for this row:

```
nm0000027  Alec Guinness  1914  2000  actor,soundtrack,writer  ... snipped ...
```

it seems that Alec Guinness is an actor, writer, and (creates) soundtract. You should create a table for all professions. You can name this table `profession`. The last field of `name.basics.tsv` is a csv value of ids of the movies that the person is known for. This also requires a table. In general, any data field in the tsv field that is a described as an "array" implies that you need to create an extra table. For these new tables, you can choose any reasonable name you like.

Here are some comments:

- Some of the names and titles uses unicode. For this assignment, we'll just pretend we are using ASCII strings. So when you use your phpmyadmin to view the data in your database, some of these strings might not display correctly.

- Recall that you can use the parameter substitution provided by Python's MySQLdb module. Since you are doing this on your local machine, there's no danger of any SQL injection attack. So you can write your own full SQL statements without using MySQLdb parameter substitution. Note that if you do use MySQLdb's parameter substitution, MySQLdb will perform suitable translation for you. For instance if an argument is python's `None` value, then it is translated as `NULL` in the SQL statement. By the way it's REALLY important to know that a MySQL `NULL` value in a column is not the same as a MySQL string `"NULL"` value!!! MySQLdb also attempts to translate Python date and datetime values into date and datetime strings during parameter substitutions.

The following python code is provided. Study it carefully. It reads `name.basics.tsv` and converts a line of data in the text file to a python list of values where each value is a string or `None` or a list of strings. The values of the professions field or all lines in the file is collected together in variable `professions` and is printed after the whole file is scanned. The python code has variable `list_indices` that allows you to specify the indices of the fields (for every row) where the value should be a python list. For instance in the code `list_indices` is set to `[4, 5]` because the values at index 4 and 5 correspond to the list of primary professions and list of title ids that the person is known for. The variable `num_fields` specifies how many fields are present in each line of text. This is used by the code to perform a minimal check that there's no data corruption such as a line with less than 6 fields or more than 6 fields. You can then manually correct that line of text or remove it completely.

```
def cleanup(x, islist=False):
    x = x.replace('\n', '')
    if x == r'\N':
        if islist:
            x = []
        else:
            x = None
    elif islist:
        x = x.split(',')
        if x == ['']: x = []
    return x

#==============================================================================
# Demonstrates how to read name.basics.tsv and for each line of text, convert
# it into a list of values where each value is a string or list of strings. The
# list is printed.
#==============================================================================
filename = 'name.basics.tsv'
list_indices = [4, 5] # indices where the value should be a list
num_fields = 6        # number of fields

f = file(filename, 'r')
f.readline() # ignore first line
professions = set([]) # professions as a set

count = 0
for line in f:

    count += 1
    if count % 1000000 == 0: print count

    items = line.split('\t')
    newitems = []
    for index, item in enumerate(items):
        if index not in list_indices:
            newitems.append(cleanup(item))
        else:
            newitems.append(cleanup(item, islist=True))
    items = newitems

    if len(items) != num_fields:
        print "This row does not have", num_fields, "fields:", items
        raise Exception

    try:
        professions |= set(items[4])
    except Exception as e:
        print "Exceptiion:", e
        print "items:", items

    print items

professions = list(professions) # convert professions to a list
print "professions", professions
```

Note that a person with his/her name in the `name` table has one or more professions. A row in the `profession` might be a profession of several people in the `name_basic` table. So this an example of a many-to-many relation. You will need a table to link these two tables. You can call this table `name_basic_profession`.

You want to plan carefully which table you want to construct first – if table T_1 references T_2, then of course T_2 has to be constructed first. Some tables can be constructed at the same time.

YOUR SQL QUERIES AND THE BACON NUMBER

Next, answer Q1-Q9 by writing SQL statements. If you believe it's impossible to answer the query, write "`select "IMPOSSIBLE"`" as your SQL statement.

Using a word processor (say LibreOffice) and create a document for all your queries and their outputs. Start the answer of each question on a new page. Make sure you indicate the question number. Print your document. Staple them together and turn in your hardcopy.

Also, email your work to `yliow.submitgmail.com` with subject "`ciss430 a03`". This should be a tar-gzipped file of your directory `a03`. In a03 there's a directory `a03q01`, etc. In `a03q01`, there a `a03q01.sql` which contains your SQL query for Q1 of `a03`. You must email with your college account.

You will need the following information to answer the queries:

The **Bacon number** of an actor/actress is the number of degrees of separation he/she has from actor Kevin Bacon, as defined by the game known as Six Degrees of Kevin Bacon. It applies the Erdös number concept (`https://en.wikipedia.org/wiki/Erd%C5%91s_number`) to the movie industry. The higher the Bacon number, the farther away from Kevin Bacon the actor is.

For example, Kevin Bacon's Bacon number is 0. If an actor worked in a movie with Kevin Bacon, the actor's Bacon number is 1. If an actor X worked with an actor Y in movie M_1 and Y who worked with Kevin Bacon in a movie M_2, then X's Bacon number is 2 (or less since X might have worked in another movie M_3 starring Kevin Bacon), and so forth.

An actor/actress n_1 has an finite Bacon number if it is possible to find a sequence of actors/actresses $n_2, ..., n_k$ such that n_i, n_{i+1} acted in the same movie and n_k is Kevin Bacon. Otherwise that actor/actress has an infinite Bacon number.

Q1. Describe of all your tables (using `desc`).

Q2. Write a query that prints the names of all the movies starring Kevin Bacon as an actor. The output must show the year of the movie, the primary title of the movie, the original title of the movie, the name of the character Kevin Bacon played. The report from the query must be sorted in descending order by date and then ascending by primary title. [WARNING: Do not include, for instance, movies where Kevin Bacon is a director.]

Q3. How many actors and actresses appear in more movies than Kevin Bacon?

Q4. Print the names of all actors and actresses with Bacon number 1. This should be sorted in ascending order. There should be no repeats. (The list is long – use a smaller font.)

Q5. Print the number of actors and actresses with Bacon number 1.

Q6. Print the number of actors and actresses with Bacon number 0 or 1 or 2.

Q7. What is the number and percentage of actors and actresses with Bacon number 0 or 1 or 2 or 3?

Q8. Excluding actors and actresses who passed away before 1978, what is the number and percentage of actors and actresses with Bacon number of 0 or 1 or 2 or 3 or 4?

Q9. Excluding actors and actresses who passed away before 1978, what is the percentage of actors and actresses with finite Bacon number?

Notes

As an example on creating the database, if you look at the `title_basics` schema below,

```
title.basics.tsv.gz - Contains the following information for titles:
tconst (string) - alphanumeric unique identifier of the title
titleType (string) - the type/format of the title (e.g. movie, short, tvseries, tvepisode, video, etc)
primaryTitle (string) - the more popular title / the title used by the filmmakers on promotional materials
                        at the point of release
originalTitle (string) - original title, in the original language
isAdult (boolean) - 0: non-adult title; 1: adult title
startYear (YYYY) - represents the release year of a title. In the case of TV Series, it is the series start
                   year
endYear (YYYY) - TV Series end year. ‘\N’ for all other title types
runtimeMinutes - primary runtime of the title, in minutes
genres (string array) - includes up to three genres associated with the title
```

Most of the fields are pretty simple. The only ones which are significant are `tconst` which is clearly a primary key and `genres` which is a non-atomic value since it's an array. You should ignore `genres` for now, and as an experiment, write the python code to build the table for all the other fields first to get a feel for this table. Along the way you have to decide for instance the maximum length of the strings. After building the table, you can view it using phpmyadmin. You probably want to just create 5 rows first. So in the python code that loops over all lines of the file, include a counter and break the loop once the counter is ≥ 5. Once it works, it's time to think about how to modify your code to handle genres.

Now for the `genres`. You need a new table, say genres. It's very simple, it's just

```
genres(id, name)
```

where `id` is a primary key - you can choose `auto_increment int` for instance. `genres.name` is just a string. You then need to link `titles_basic` and `genres`. You can create a table

```
titles_basic_genres(id, tconst, genres_id)
```

(The `id` is not really necessary - you can omit that if you wish.) Clearly `titles_basic_genres.tconst` references `titles_basic.tconst` and `titles_basic_genres.genres_id` references `genres.id`.

You can now drop the original `title_basic` table, rewrite your python code to create rows for `title_basic`, and for each `title_basic` row created, look at its genres. Loop over the genres for that title, see if the genre is in the genre table. If it's found, link this `title_basic` with the genre row using `titles_basic_genres`. If the genre is not found, create it first before linking. Again you want to have a counter and stop adding rows once counter ≥ 5. Check your tables to make sure your code is working. Run your program again to get more rows, say 100, and check your tables. Once you are confident your python code is doing the right thing, remove the counter in your code and run through all the movie titles.

You would expect the genres table to be very small. You should visually browse through the table and see if there are any data corruption. To prevent data corruption on your side, you should use constraints as much as possible. For instance genres.name should be unique and not null.

Once you are done with the above, you will see that the process for creating most of the other tables are the same. You want to spend some time tidying up your code for the above since it can be re-used again in another table.

The process of incorporating data from various sources is very common in the industry.

Once you are done with all dataset, tidy up your code completely and you can create an open source project for this assignment. You can run a massive data analysis on ALL actors and actresses and for each compute their average number (just like for "Kevin Bacon number", you can also have "Sean Connery number") and see who are the "centers of Hollywood". For that, it's better that you first build an adjacency matrix for all actors and actresses as a sparse matrix (see ciss350 notes). The schema is just

```
matrix(nconst1, nconst2)
```

If the actor/actress with `nconst1` as primary key in the `name.basic` table and the actor/actress with `nconst2` as primary key appear in the same movie, then `(nconst1, nconst2)` appears in `matrix`. If `(nconst1, nconst2)` did not appear in the same movie, then they do not appear in this table. Finding the shortest path between every pair of actors/actresses is similar to the process of computing the transitive closure of a relation.

The above is a very important idea and extremely valuable in the industry. It's used by facebook (social network graphs), uber, google, mapping, airflight planning, etc. So this is assignment is NOT just an academic exercise.

ASIDE

- The database is huge. First focus only on the tables you need to answer all questions. After that, finish the remaining tables.
- When it comes to building a database from a dataset (or multiple datasets), it's possible to have data corruption or unexpected data formats. In case the program that builds the database halts, you need to rerun it again. You want to write the program such that when you restart it you need to fully rebuild the database. Find a way to restart the build process at different points in the process.
- The database is huge. For string data that will go into a varchar column, you need to know the size of that varchar column. Therefore you should run through the datasets and figure out the length of the different fields first before creating the tables.

- Make sure you have enough disk space. Like I said this is a *huge* database. Use the linux command `df` to figure out how much space you have available.