Assignment 1 Introduction to SQL

Brief

- **Due date**: 21st September, 2016, 11:59 pm
- **Data**: HW1-sql.zip (social.db, matrix.db, university.db, sql_python_demo.py) posted on Blackboard
- **Hand in**: follow the hand in instruction

Tools

SQLite3

In this assignment you are required to use SQLite3 on your Azure virtual machine or your local Unix like machine to write and execute SQL statements. To install SQLite3 on the VMs, simply run:

```
$ sudo apt-get install sqlite3
```

- Download the attached "HW1-sql.zip" file.

- Unzip the files in a directory (e.g `../data`) in your VM. For MAC/Linux users, you can use "scp" or "sftp" command to transfer files between your laptop and the VM. Windows users can use "winscp".

- Use an editor to open **sql_python_demo.py** file and finish all the marked ***TODO*** to finish all the questions related to sqlite. Learning how to use **DB-API 2.0 interface for SQLite databases** is also part of the purpose of this homework. Please refer to the "Handing in" section on this write-up for more information.

SQLite3 can also be accessed from the command line using `sqlite3`. Although we do not use sql file for submission, you could still try this way of do operation on database.

To load a pre-existing database, you can run the following command:

```
$ sqlite3 ../data/[database_file].db
```

Now, you may enter a SQL statement in front of `sqlite3>` and execute it using a ";" at the end.

For more information on using SQLite from the command line, see

http://www.sqlite.org/sqlite.html.

Problem 1: Social Network Analysis

We provided a simple database for a social network, `social.db`. Here's the schema:

```
Student(ID int, name text, grade int)
Friend(ID1 int, ID2 int)
Likes(ID1 int, ID2 int)
```

In the **Student** table, **ID** is a unique identifier for a particular student. **name** and **grade** correspond to the student's first name and grade.

In the **Friend** table, each (**ID1**, **ID2**) pair indicates that the student with **ID1** is friends with the student with **ID2**(and vice versa). The friendship is mutual, and if (**ID1**, **ID2**) is in the table, it is guaranteed that (**ID2**, **ID1**) exists in the table.

In the **Likes** table, each (**ID1**, **ID2**) pair indicates that the student with **ID1** likes the student with **ID2**. The (**ID1**, **ID2**) pair in the table does not guarantee that the (**ID2**, **ID1**) pair also exists in the table.

1. Write a SQL statement that returns the names of students who are in grade 9. Results should be ordered by name (A-Z). Write **query_1** in provided python script.

2. Write a SQL statement that returns the number of students in each grade. The query should return 2 columns: grade, count. Results should be ordered by grade (ascending). Write **query_2** in provided python script.

3. Write a SQL statement that returns the name and grade of all students who have more than 2 friends. Results should be ordered by name (A-Z), then grade (ascending). Write **query_3** in provided python script.

4. Write a SQL statement that returns the name and grade of all students who are liked by at least one student who is in any grade above. Results should be ordered by name (A-Z), then grade (ascending). Write **query_4** in provided python script.

5. Write a SQL statement that returns the name and grade of all students who only like their friends. You should also return students who don't like anyone as well. Results should be ordered by name (A-Z), then grade (ascending). Write **query_5** in provided python script.

6. Write a SQL statement to find pairs (A,B) such that student A likes student B, but A is not friends with B. The query should return 4 columns: ID1, name1, ID2, name2. Results should be ordered by ID1 (ascending), then ID2 (ascending). Write **query_6** in provided python script.

7. **(Extra Credit)** For each pair (A,B) in the previous problem, find all the students C who can introduce them (C is friends with both A and B). The query should return 6 columns: ID1, name1, ID2, name2, ID3, name3. Results should be ordered by ID1 (ascending), ID2 (ascending), then ID3 (ascending). Write **query_7** in provided python script.

Problem 2: Course Evaluation Data Analysis

A group of anonymous hackers have leaked the course evaluation data of the CS department at Grey University (our long time rival!). Turns out, they were using the Star schema for their database:

```
Fact_Course_Evaluation (

    professor_id int,
    term_id int,
    student_id int,
    type_id int,
    class_score int,
    efficiency_score int,
    content_score int
)


Dim_Term (

    id int,
    year int,
    term text (possible values are 'spring' and 'fall')
)


Dim_Student (

    id int,
```

```
    name text,
    college_year_name text, (possible values are 'freshman', 'sophomore', 'junior',
'senior')

    undergraduate bit

)


Dim_Professor (

    id int,
    name text,
    gender text, (possible values are 'male' and 'female')
    tenured bit

)


Dim_Type (

    id int,
    type text, (possible values are 'course', 'seminar' and 'independent_study')
    area text (possible values are 'ai', 'systems', 'theory')
)
```

The database is provided as university.db.

1. Write a SQL statement to find out if tenured professors get better class scores than untenured professors. The query should return the average class scores for tenured and untenured professors. Write **query_9** in provided python script. Make sure your output is in format:

```
<q9>
(0, [average class score])
(1, [average class score])
</q9>
```

2. Write a SQL statement to find the average class scores for all courses, grouped by the area over the different years. The query should return 3 columns: year, area, avg_score. Results should be ordered by year (ascending), then area (A-Z). Write **query_10** in provided python script.

3. If classes can belong to more than one area (e.g., it can be both a theory and systems class), what changes to the database schema are necessary? Save your answer to **[your netid].txt**

Problem 3: Sparse Matrix Multiplication

A sparse matrix is a matrix populated primarily with zeros as elements of the table. Systems designed to efficiently support sparse matrices look a lot like databases: The **(i, j)**entry of the matrix is represented as a record **(i, j ,value)**in the database. The benefit is that you only need one record for every non-zero element in the matrix. For example, the matrix:

0	2	0
1	0	0
0	0	-3
0	0	0

can be represented as a table:

rc	column #	valu e
0	1	2
1	0	1
2	2	-3

so that instead of having to store 12 integers, we only need to store 9.

Now, since you can represent a sparse matrix as a table, it's reasonable to consider whether you can express matrix multiplication as a SQL query.

In matrix.db, there are two matrices **A**and **B**represented as follows:

A(**row_num** int, **col_num** int, **value** int)

B(**row_num** int, **col_num** int, **value** int)

The matrix **A**and matrix **B**are both square matrices with 5 rows and 5 columns.

Express A x B as a SQL query. The query should return 3 columns: row_num, col_num, value. Results should be ordered by row_num (ascending), then col_num (ascending). Write **query_11** in provided python script.

If you're wondering why this might be a good idea, consider that advanced databases execute queries in parallel. So it can be quite efficient to process a very large sparse matrix in a database (millions of rows and columns)!

Handing in

The provided python script **sql_python_demo.py** gives you a simple demonstration for how to access sqlite3 using python API. For more instructions on using python sqlite3 API, please visit: https://docs.python.org/2/library/sqlite3.html. Learning how to use **DB-API 2.0 interface for SQLite databases**is also part of the purpose of this homework.

First of all, change **sql_python_demo.py** into **[your NetID].py**

Then You need to read and understand script. Run the script, you will see that it will output result for the query_1 at file **demo.result**, with extension **.result** as below:

```
<q1>
(16,)
</q1>
```

Finish all the marked ***TODO*** to finish all the questions related to sqlite. Run your Python Script and make sure that the final output file is

[your_netid].result and the format is like:

```
<q1>
answer for query 1
</q1>
<q2>
answer for query 2
</q2>
<q3>
answer for query 3
</q3>
...
```

all answers should be surrounded by tag like **<q1> … </q1>,** with no blank lines in between. As long as you do not change the function **def output_result(index, result),** everything should be just fine.Note that, Python willoutput result not like the result when you do using command line tool **sqlite3** for writing SQL

directly. For instance it might output a Python tuple **(u'Tiffany',)** instead of 'Tiffany', which is totally fine. The tuple format is exactly what we need, the autograder will be used to compare with the correct result, which is also in tuple format.

Your answer will be auto graded based on the output file of your Python script.

However, the sql statement in Python script will also be reviewed manually.

For the 3nd question of problem 2, you need to write down your answer and save it into a txt file -- **[your NetID].txt**

For submitting to blackboard, create a folder with your NetID, put only two file

[your NetID].py and [your NetID].txt in the folder and zip it.

You **MUST** zip the folder into **[your NetID].zip**, not other file extension.

Late submission is not acceptable. The submission link in Blackboard will disappear immediately at the deadline. You could submit multiple times, only the latest version will be graded.