**DSC 450: Database Processing for Large-Scale Analytics**

**Assignment Module 9**

**Part 1**

Using the same tweets from: http://rasinsrv07.cstcis.cti.depaul.edu/CSC455/Assignment5.txt

Create a third table, now incorporating the Geo table (in addition to tweet and user tables that you already have) and extend your schema accordingly. You do not need to use ALTER TABLE, it is sufficient to just re-make your schema.

You will need to generate an ID for the Geo table primary key (you may use any value or reasonable combination of values as long as it is unique) for that table and link it to the Tweet table (foreign key should be in the Tweet). However, that value should be related to the location (i.e., same value for same location) – do not use a simple incremental key.

In addition to the primary key column, the geo table should have the “type”, “longitude” and “latitude” columns. **NOTE**: that there are no longitude/latitude dictionary keys. Instead, the geo dictionary has a 2-value tuple which are the lon/lat coordinates.

Load the tweet data into your new tables.

﻿import urllib

import json

import sqlite3

def createDict(url):

conn = sqlite3.connect('dsc450\_HW9\_2.db')

c = conn.cursor()

conn.commit()

webFD = urllib.request.urlopen(url)

x = webFD.readlines()

tweets = '''CREATE TABLE UserDict

(

id VARCHAR2(45),

name NUMBER(20),

screen\_name VARCHAR2(140),

description VARCHAR2(75),

friend\_count VARCHAR2(25),

user\_id VARCHAR2(25),

CONSTRAINT userDict\_PK

PRIMARY KEY(id, user\_id),

Foreign Key (user\_id)

REFERENCES tweet1(id\_str)

); '''

c.execute(tweets)

other\_table = '''CREATE TABLE tweet1

(

created\_at VARCHAR2(45),

id\_str VARCHAR2(25),

text VARCHAR2(140),

source VARCHAR2(75),

in\_reply\_to\_user\_id VARCHAR2(25),

in\_reply\_to\_screen\_name VARCHAR2(25),

in\_reply\_to\_status\_id VARCHAR2(25),

retweet\_count NUMBER(5),

contributors VARCHAR2(25),

CONSTRAINT tweet\_PK

PRIMARY KEY(id\_str, text)

); '''

c.execute(other\_table)

geo\_table = '''CREATE TABLE geo

(

PK VARCHAR(45),

type VARCHAR2(6),

longtitude VARCHAR(30),

latitude VARCHAR2(30),

user\_id VARCHAR2(25),

CONSTRAINT Geo\_PK

PRIMARY KEY(PK),

Foreign Key (user\_id)

REFERENCES tweets1(id\_str)

); '''

c.execute(geo\_table)

for line in x:

try:

tweets1 = json.loads(line.decode('utf8'))

tweetID = tweets1['user']['id']

created\_at = tweets1['created\_at']

name = tweets1['user']['name']

screen\_name = tweets1['user']['screen\_name']

description = tweets1['user']['description']

friends\_count = tweets1['user']['friends\_count']

user\_id = tweets1['id\_str']

tweetList = [tweetID, name, screen\_name, description, friends\_count, user\_id]

c.executemany("INSERT INTO UserDict VALUES(?,?,?,?,?,?)", (tweetList,))

text = tweets1['text']

source = tweets1['source']

replyuserid = tweets1['in\_reply\_to\_user\_id']

replyscreenname = tweets1['in\_reply\_to\_screen\_name']

replystatusid = tweets1['in\_reply\_to\_status\_id']

retweet = tweets1['retweet\_count']

constributors = tweets1['contributors']

tweetList1 = [created\_at, user\_id, text, source, replyuserid, replyscreenname, replystatusid, retweet, constributors]

c.executemany("INSERT INTO tweet1 VALUES(?,?,?,?,?,?,?,?,?)", (tweetList1,))

geo = tweets1['geo']

if geo != 'NULL':

longLat = tweets1['geo']['coordinates']

long = longLat[0]

lat = longLat[1]

pk = long + lat

tweetList2 = [pk, geo, long, lat, user\_id]

c.executemany("INSERT INTO geo VALUES(?,?,?,?,?)", (tweetList2,))

except ValueError:

file = open('Module7\_errors.txt', 'w')

tweet = line.decode()

file.write(tweet)

createDict('http://rasinsrv07.cstcis.cti.depaul.edu/CSC455/Assignment5.txt')

* 1. Write and execute a SQL query to do the following. Time and report the runtime of your query.

Find tweets where tweet id\_str contains “777” or “88” anywhere in the column\

DECLARE @t1 DATETIME;

DECLARE @t2 DATETIME;

SET @t1 = GETDATE();

SELECT \* FROM tweet1 WHERE id\_str LIKE ‘%777%’ OR LIKE ‘%88%’;

SET @t2 = GETDATE ();

SELECT DATEDIFF (millisecond,@t1, @t2) AS elapsed\_MS;

* 1. Write the equivalent of the previous query in python (without using SQL) by reading it from the file. Time and report the runtime of your query.

﻿import time

start = time.time()

conn = sqlite3.connect('dsc450\_HW9\_2.db')

c = conn.cursor()

conn.commit()

query1 = """ SELECT \* FROM tweet1 WHERE id\_str LIKE (%777%) OR LIKE (%88%); """

c.execute(query1)

end = time.time()

print('Elapsed time: ' + str(end - start) + ' seconds')

* 1. Write and execute a SQL query to do the following. Time and report the runtime of your query.

Find how many unique values are there in the “in\_reply\_to\_user\_id” column

DECLARE @t1 DATETIME;

DECLARE @t2 DATETIME;

SET @t1 = GETDATE();

SELECT COUNT(DISTINCT in\_reply\_to\_user\_id from tweet1);

SET @t2 = GETDATE ();

SELECT DATEDIFF (millisecond,@t1, @t2) AS elapsed\_MS;

* 1. Write the equivalent of the previous query in python (without using SQL) by reading it from the file. Time and report the runtime of your query.

﻿import time

start = time.time()

conn = sqlite3.connect('dsc450\_HW9\_2.db')

c = conn.cursor()

conn.commit()

query1 = """ SELECT \* FROM tweet1 WHERE id\_str LIKE '%777%' OR id\_str LIKE '%88%'; """

c.execute(query1)

end = time.time()

print('Elapsed time: ' + str(end - start) + ' seconds')

The reported runtime was recored as: ﻿Elapsed time: 0.00909113883972168 seconds

* 1. Use python to plot the lengths of first 40 tweets (only 40, not all of the tweets) versus the length of the username for the user on a graph. Create a scatterplot. Submit both your python code and the resulting graph file.

﻿import urllib

import json

import matplotlib.pyplot as plt

def createDict(url):

fig = plt.figure()

sp1 = fig.add\_subplot(2,2,1)

webFD = urllib.request.urlopen(url)

x = webFD.readlines()

counter = 0

tweetLength = []

userNameLength = []

for line in x:

try:

tweets1 = json.loads(line.decode('utf8'))

while counter <41:

counter = counter + 1

x = len(tweets1['text'])

y = len(tweets1['user']['name'])

tweetLength.append(x)

userNameLength.append(y)

except ValueError:

file = open('Module7\_errors.txt', 'w')

tweet = line.decode()

file.write(tweet)

print(tweetLength, userNameLength)

sp1.scatter(tweetLength, userNameLength)

createDict('http://rasinsrv07.cstcis.cti.depaul.edu/CSC455/Assignment5.txt')

I think I may have made an error, because the only values returned for text length are 113, and the only length returned for username length is 10.

![Chart, scatter chart

Description automatically generated]()

**Part 2**

1. Create an index on userid in Tweet table in SQLite (submit SQL code for this question). These questions are as straightforward as they look, you just need to create an index.

CREATE INDEX TweetIndex ON tweet1(user\_id);

1. Create a composite index on (friends\_count, screen\_name) in User table (submit SQL code for this question)

CREATE INDEX UserIndex ON UserDict(friend\_count, screen\_name);

1. Create a materialized view (using CREATE TABLE AS because SQLite does not have full support for MVs) that answers the query in Part-1-a. Submit your SQL code.

CREATE TABLE AS TimeTable AS SELECT \* FROM tweet1 WHERE id\_str LIKE ‘%777%’ OR LIKE ‘%88%’;

**Part 3**

1. Draw a precedence graph.

**T1 T2 T3**

**W(A)**

**W(A)**

**W(C)**

**R(E)**

**W(B)**

**W(C)**

**The only conflict that exists is between W(A) for transactions 1 and 3, so we cannot swap those rows. But we are able to move the other items around as much as necessary.**

Is the schedule serializable? If not, say “no”. If it is, name at least one equivalent serial schedule (e.g., <T1, T2, T3> if the execution is equivalent to individual operations executing in that order)

|  |  |  |
| --- | --- | --- |
| T1 | T2 | T3 |
|  |  | W(A) |
| W(A) |  |  |
|  | W(C) |  |
|  |  | R(E) |
| W(B) |  |  |
|  | W(C) |  |

**Yes this schedule is serializable. The only time when two different transactions read or write the same data element is with the first action of transaction 1 and transaction 3. Since transaction 3 writes A before transaction 1, we need to keep that action first. To change this schedule so that it is serializable, I would created the following schedule:**

|  |  |  |
| --- | --- | --- |
| T1 | T2 | T3 |
|  |  | W(A) |
|  |  | R(E) |
| W(A) |  |  |
| W(B) |  |  |
|  | W(C) |  |
|  | W(C) |  |

**We can move the reading of element E up, because no other transaction reads or writes element E. We can also move the writing of element B for transaction 1 higher, because no other transaction reads or writes element B. And finally, we can move the first write of C down, because no other transaction works with element C.**

1. Draw a precedence graph.

**T1 T2 T3**

**R(X)**

**R(X)**

**W(X)**

**R(X)**

**W(X)**

**W(X)**

**There are two main conflicts between transaction 1and 3. And because we cannot change the other without changing the operations this schedules is not serializable.**

Is the schedule serializable? If not, say “no”. If it is, name at least one equivalent serial schedule (e.g., <T1, T2, T3> if the execution is equivalent to individual operations executing in that order)

|  |  |  |
| --- | --- | --- |
| T1 | T2 | T3 |
|  |  | R(X) |
|  | R(X) |  |
|  |  | W(X) |
| R(X) |  |  |
| W(X) |  |  |
|  |  | W(X) |