CS 564 Problem Set #1

January 31, 2018

DELIVERABLES

Submit your queries (and only those) using the `submission_template.txt` file that is posted on the class website. Follow the instructions on the file! Upload the file at Canvas (under PS1).

INSTRUCTIONS / NOTES

- Using the IPython version of this problem set is strongly recommended, however you can use only this PDF to do the assignment, or replicate the functionality of the IPython version by using this PDF + your own SQLite interface
- Some of the problems involve *changing* this database (e.g. deleting rows)- you can always re-download `PS1.db` or make a copy if you want to start fresh!

1 Problem 1: Linear Algebra [25 points]

Two random 3x3 ($N = 3$) matrices have been provided in tables `A` and `B`, having the following schema:

- `i INT`: Row index
- `j INT`: Column index
- `val INT`: Cell value

Note: all of your answers below *must* work for any *square* matrix sizes, i.e. any value of N.

Note how the matrices are represented- why do we choose this format? Run the following queries to see the matrices in a nice format:

1.1 Part (a): Matrix transpose [5 points]

Transposing a matrix A is the operation of switching its rows with its columns- written A^T. For example, if we have:

$$A = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

Then:

$$A^T = \begin{bmatrix} a & d & g \\ b & e & h \\ c & f & i \end{bmatrix}$$

Write a *single SQL query* to get the matrix transpose A^T (in the same format as A- output tuples should be of format `(i,j,val)` where `i` is row, `j` is column, and the output is ordered by row then column index)

The output should be as follows:

```
Out[]: [(0, 0, 7),
        (0, 1, 10),
        (0, 2, 2),
        (1, 0, 5),
        (1, 1, 7),
        (1, 2, 0),
        (2, 0, 8),
        (2, 1, 7),
        (2, 2, 5)]
```

1.2 Part (b): Dot product I [5 points]

The *dot product* of two vectors $a = \begin{bmatrix} a_1 & a_2 & \dots & a_n \end{bmatrix}$ and $b = \begin{bmatrix} b_1 & b_2 & \dots & b_n \end{bmatrix}$ is

$$a \cdot b = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$

Write a *single SQL query* to take the dot product of the **second column of** A and the **third column of** B**.** The output should be:

```
Out[]: [(113,)]
```

1.3 Part (c): Dot product II [5 points]

Write a *single SQL query* to take the dot product of the **second *row* of** A and the **third column of** B**.** The output should be:

```
Out[]: [(212,)]
```

1.4 Part (d): Matrix multiplication [10 points]

The product of a matrix A (having dimensions $n \times m$) and a matrix B (having dimensions $m \times p$) is the matrix C (of dimension $n \times p$) having cell at row i and column j equal to: $C_{ij} = \sum_{k=1}^{m} A_{ik} B_{kj}$.

In other words, to multiply two matrices, get each cell of the resulting matrix C, C_{ij}, by taking the *dot product* of the ith row of A and the jth column of B.

Write a single SQL query to get the matrix product of A and B (in the same format as A and B). The output should be:

```
Out[]: [(0, 0, 106),
        (0, 1, 80),
        (0, 2, 171),
        (1, 0, 146),
```

```
(1, 1, 109),
(1, 2, 212),
(2, 0, 23),
(2, 1, 17),
(2, 2, 55)]
```

2 Problem 2: The Sales Database [35 points]

We've prepared and loaded a dataset related to sales data from a company. The dataset has the following schema:

```
Holidays (WeekDate, IsHoliday)

Stores (Store, Type, Size)

TemporalData(Store,WeekDate,Temperature,FuelPrice,CPI,UnemploymentRate)

Sales (Store, Dept, WeekDate, WeeklySales)
```

Before you start writing queries on the database, find the schema and the constraints (keys, foreign keys).

2.1 Part (a): Sales during Holidays [5 points]

Using a *single SQL query*, find the stores with the largest and smallest overall sales during holiday weeks. Further requirements:

- Use the `WITH` clause before the main body of the query to compute a subquery if necessary.
- Return a relation with schema `(Store, AllSales)`.

The output should be:

```
Out[]: [(20, 22490350.81000001), (33, 2625945.1900000004)]
```

2.2 Part (b): When Holidays do not help Sales [10 points]

Using a *single SQL query*, compute the **number** of non-holiday weeks that had larger sales than the overall average sales during holiday weeks. Further requirements:

- Use the `WITH` clause before the main body of the query to compute a subquery if necessary.
- Return a relation with schema `(NumNonHolidays)`.

The output should be:

```
Out[]: [(8,)]
```

2.3 Part (c): Total Sales [5 points]

Using a *single SQL query*, compute the total sales per month overall for each type of store. Further requirements: Return a relation with schema `(type, Month, TotalSales)`.

Hint: SQLite3 does not support native operations on the DATE datatype. To create a workaround, you can use the `LIKE` predicate and the string concatenation operator (`||`). You can also use the substring operator that SQLite3 supports (`substr`).

The output should be:

```
Out[]: [('A', '01', 214176168.09999868),
        ('A', u'02', 366507672.3800041),
        ('A', '03', 380774533.0999994),
        ('A', '04', 416180129.8799989),
        ('A', '05', 359086595.83999956),
        ('A', '06', 399448005.7700002),
        ('A', '07', 417243210.38999707),
        ('A', '08', 394863683.68999743),
        ('A', '09', 373118622.1900005),
        ('A', '10', 377131480.28000176),
        ('A', '11', 264721386.60999998),
        ('A', '12', 367763234.5200006),
        ('B', '01', 95446454.79999965),
        ('B', '02', 167671723.73999923),
        ('B', '03', 175136037.61000007),
        ('B', '04', 190880594.58999932),
        ('B', '05', 163455609.2500001),
        ('B', '06', 186362500.36000022),
        ('B', '07', 193743231.9999991),
        ('B', '08', 181504990.0400001),
        ('B', '09', 168953941.7599988),
        ('B', '10', 170604194.5999999),
        ('B', '11', 125545696.86999977),
        ('B', '12', 181395761.19999966),
        ('C', '01', 22975815.59),
        ('C', '02', 34548494.330000006),
        ('C', '03', 36875330.56999996),
        ('C', '04', 39799060.49999998),
        ('C', '05', 34583366.86000001),
        ('C', '06', 36819380.709999934),
        ('C', '07', 39014534.860000074),
        ('C', '08', 36721535.09000013),
        ('C', '09', 36688615.31000005),
        ('C', '10', 37049112.70999994),
        ('C', '11', 22748641.530000072),
        ('C', '12', 27679639.47999999)]
```

2.4 Part (d): Computing Correlations [15 points]

The goal of this exercise is to find out whether each one of the 4 numeric attributes in `TemporalData` is positively or negatively correlated with sales.

For our purposes, the intuitive notion of correlation is defined using a standard statistical quantity known as the *Pearson correlation coefficient*. Given two numeric random variables X and Y, it is defined as follows:

$$\rho_{X,Y} = \frac{E[XY] - E[X]E[Y]}{\sqrt{E[X^2] - E[X]^2}\sqrt{E[Y^2] - E[Y]^2}}$$

On a given sample of data with n examples each for X and Y (label them x_i and y_i respectively for $i = 1 \dots n$), it is estimated as follows:

$$\rho_{X,Y} = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}$$

In the above, $\bar{x}$ and $\bar{y}$ are the sample means, i.e., $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$, and $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} y_i$.
Further requirements:

- Return a relation with schema `(AttributeName VARCHAR(20), CorrelationSign Integer)`.
- The values of AttributeName can be hardcoded string literals, but the values of `CorrelationSign` must be computed automatically using SQL queries over the given database instance.
- You can use multiple SQL statements to compute the result. It might be of help to create and update your own tables.

The output should be:

```
Out[]: [(u'Temperature', -1),
        (u'FuelPrice', -1),
        (u'CPI', -1),
        (u'UnemploymentRate', -1)]
```

3 Problem 3: The Traveling SQL Server Salesman Problem [40 points]

SQL Server salespeople are lucky as far as traveling salespeople go- they only have to sell one or two big enterprise contracts, at one or two offices in Wisconsin, in order to make their monthly quota!

Answer the following questions using the table of streets connecting company office buildings.

Note that for convenience all streets are included *twice*, as $A \to B$ and $B \to A$. This should make some parts of the problem easier, but remember to take it into account!

3.1 Part (a): One-hop, two-hop, three-hop... [10 points]

Our salesperson has stopped at UW-Madison, to steal some cool new RDBMS technology from CS564-ers, and now wants to go sell it to a company _within 10 miles of UW-Madison and *passing through no more than 3 distinct streets*. Write a single query, not using `WITH` (see later on), to find all such companies.

Your query should return the schema `(company, distance)` where distance is cumulative from UW-Madison. The output should be:

```
Out[]: [(u'DooHickey Collective', 7),
        (u'DooHickey Corp', 9),
        (u'Gadget Collective', 9),
        (u'Gadget Corp', 6),
        (u'Gizmo Corp', 9),
        (u'Widget Industries', 10)]
```

3.2 Part (b): A stop at the Farm [10 points]

Now, our salesperson is out in the field, and wants to see all routes- and their distances- which will take him/her from a company A to a company B, with the following constraints:

- The route must _pass through UW-Madison (in order to pick up new RDBMS tech to sell!)
- A and B must *each individually* be _within 3 hops of UW-Madison
- A and B must be different companies
- *The total distance must be* $<= 15$
- Do not use `WITH`
- If you return a path $A \rightarrow B$, *also include* $B \rightarrow A$ *in your answer*

In order to make your answer a bit cleaner, you may split into two queries, one of which creates a `VIEW`. A view is a virtual table based on the output set of a SQL query. A view can be used just like a normal table- the only difference under the hood is that the DBMS re-evaluates the query used to generate it each time a view is queried by a user (thus the data is always up-to date!)

Here's a simple example of a view:

```
DROP VIEW IF EXISTS short_streets;
CREATE VIEW short_streets AS
SELECT A, B, d FROM streets WHERE d < 3;
SELECT * FROM short_streets LIMIT 3;
```

The output should be:

```
Out[]: [(u'DooHickey Collective', u'Gadget Corp', 13),
        (u'DooHickey Corp', u'Gadget Corp', 15),
        (u'Gadget Collective', u'Gadget Corp', 15),
        (u'Gadget Corp', u'DooHickey Collective', 13),
        (u'Gadget Corp', u'DooHickey Corp', 15),
        (u'Gadget Corp', u'Gadget Collective', 15),
        (u'Gadget Corp', u'Gizmo Corp', 15),
        (u'Gizmo Corp', u'Gadget Corp', 15)]
```

3.3 Part (c): Ensuring acyclicity [10 points]

You may have noticed at this point that the network, or ***graph***, of streets in our `streets` table seems like it might be a **tree**.

Recall that a ***tree*** is an undirected graph where each node has exactly one parent (or, is the root, and has none), but may have multiple children. A slightly more formal definition of a tree is as follows:

> An undirected graph T is a ***tree*** if it is **connected**- meaning that there is a path between every pair of nodes- and has no **cycles** (informally, closed loops)

Suppose that we guarantee the following about the graph of companies & streets determined by the `streets` table: (i) It is *connected*, and (ii) It has no cycles of length > 3.

Write a *single SQL query* which, if our graph is not a tree (i.e. if this isn't a shaggy dog problem), **turns it into a tree** by deleting exactly *one* street (= *two entries in our table!*).

3.4 Part (d): The longest path [10 points]

In this part, we want to find the distance of the *longest path between any two companies*.

Note that you should probably have done Part (c) first so that the graph of streets is a *tree*- this will make it much easier to work with!

If you've done the other parts above, you might be skeptical that SQL can find paths of arbitrary lengths (which is what we need to do for this problem); how can we do this?

There are some non-standard SQL functions- i.e. not universally supported by all SQL DBMSs- which are often incredibly useful. One of these is the `WITH RECURSIVE` clause, supported by SQLite. A `WITH` clause lets you define what is essentially a view within a clause, mainly to clean up your SQL code. A trivial example, to illustrate `WITH`:

```
WITH companies(name) AS (
    SELECT DISTINCT A FROM streets)
SELECT *
FROM companies
WHERE name LIKE '%Gizmo%';
```

There is also a recursive variant, `WITH RECURSIVE`. `WITH RECURSIVE` allows you to define a view just as above, except the table can be defined recursively. A `WITH RECURSIVE` clause must be of the following form:

```
WITH RECURSIVE
    R(...) AS (
        SELECT ...
        UNION [ALL]
        SELECT ... FROM R, ...
    )
...
```

`R` is the *recursive table*. The `AS` clause contains two `SELECT` statements, conjoined by a `UNION` or `UNION ALL`; the first `SELECT` statement provides the initial / base case values, and the second or *recursive* `SELECT` statement must include the recursive table in its `FROM` clause.

Basically, the recursive `SELECT` statement continues executing on the tuple *most recently inserted into* `R`, inserting output rows back into `R`, and proceeding recursively in this way, until it no longer outputs any rows, and then stops. See the SQLite documentation for more detail.

The following example computes $5! = 5 * 4 * 3 * 2 * 1$ using `WITH RECURSIVE`:

```
WITH RECURSIVE
    factorials(n,x) AS (
        SELECT 1, 1
        UNION
        SELECT n+1, (n+1)*x FROM factorials WHERE n < 5)
SELECT x FROM factorials WHERE n = 5;
```

In this example:

1. First, `(1,1)` is inserted into the table `factorials` (the base case).
2. Next, this tuple is returned by the recursive select, as `(n, x)`, and we insert the result back into `factorials`: `(1+1, (1+1)*1) = (2,2)`
3. Next, we do the same with the last tuple inserted into `factorials`- `(2,2)`- and insert `(2+1, (2+1)*2) = (3,6)`
4. And again: get `(3,6)` from `factorials` and insert `(3+1, (3+1)*6) = (4,24)` back in
5. And again: `(4,24)` -> `(4+1, (4+1)*24) = (5,120)`
6. Now the last tuple inserted into `factorials` is `(5, 120)`, which fails the `WHERE n < 5` clause, and thus our recursive select returns an empty set, and our `WITH RECURSIVE` statement concludes
7. Finally, now that our `WITH RECURSIVE` clause has concluded, we move on to the `SELECT x FROM factorials WHERE n=5` clause, which gets us our answer!

Now, your turn! Write a single SQL query that uses `WITH RECURSIVE` to find the furthest (by distance) pair of companies that still have a path of streets connecting them. Your query should return `(A, B, distance)`. The output should be:

```
Out[]: [(u'GadgetWorks', u'ThingWorks', 63)]
```

NOTE: The ***distance*** of the longest path could be **49** ***OR*** **63** depending on which street you deleted in Part (c)!