CS 145 PS1

September 25, 2016

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
- Note that the problems reference tables in the SQLite database (in the PS1.db file) however solution queries must be for any general table of the specified format, and so use of the actual database provided is *not necessary*
- See Piazza for submission instructions
- Have fun!

1 Problem 1: Linear Algebra

Consider two random 3x3 ($N = 3$) matrices 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 also how the matrices are represented- why do we choose this format?

1.1 Part (a): Matrix transpose

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 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). If executed on the tables A and B in PS1.db, the result should be:

```
i j val
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

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 query to take the dot product of the **second column of** A and the **third column of** B. If executed on the tables A and B in PS1.db, the result should be 113.

1.3 Part (c): Dot product II

Write a single query to take the dot product of the **second *row* of** A and the **third column of** B. If executed on the tables A and B in PS1.db, the result should be 212.

1.4 Part (d): Matrix multiplication

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). If executed on the tables A and B in PS1.db, the result should be:

```
i j val
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: U.S. Hourly NOAA Precipitation dataset

We've prepared and loaded a public dataset (https://catalog.data.gov/dataset/u-s-hourly-precipitation-data) from the US NOAA (National Oceanic and Atmospheric Administration) of daily precipitation data from weather stations across CA from the month of Sep. 2013. We'll use the 'precipitation' table here, which has a simplified schema:

```
station_id    INT
day           INT
precipitation INT
```

Again, use of the actual data is not required to solve this problem (only your SQL query), but using it is highly recommended!

2.1 Part (a): State Champions

Using a *single SQL query*, find all of the stations that had the highest daily precipitation (across all CA stations) on any given day **more than once**, and return the counts of how many "best days" they had in descending order. Further requirements:

- Use GROUP BY
- Write the shortest possible SQL query to accomplish this
- Return relation (station_id, num_best_days)

Hint: Make sure your query correctly handles ties! If you ran your query on the actual dataset provided, you should get:

```
station_id num_best_days
335701      6
92707       3
225707      2
458701      2
611807      2
734201      2
```

2.2 Part (b.i): Median value, Pt. I

Our goal in this part is going to be to find the **median value** of a list of values. We want to do this for the general case, however we'll start with a slightly simplified setting: Write a *single SQL query* to find the median value of a certain attribute in a table, given that:

- The table contains an odd number of rows
- The values of this attribute are unique in the table (e.g. no two rows have the same value for this attribute)

Also, **do not hard code the size of the table and/or use any ORDER BY clause. Think about the definition of median!** Write your query for a table X, constructed out of the distinct precipitation values of all stations on all days, having one attribute p. If you constructed such a table and ran your SQL query on it, you should get a median value of 51.

2.3 Part (b.ii): Median value, Pt. II

Now, we want to write a *single SQL query* to find the median precipitation value across all days for a station. **Note that to get credit, your query must work for ANY station**. Again, do not hardcode the size of any table and/or use any ORDER BY clause. Note that now:

- The number of rows can be even
- The values can have duplicates

Also note that we will use the definition of median where ties (e.g. when there are an even number of rows) are broken by averaging. If you executed your query on the provided data for station 376101, you should get a median value of 15.0.

2.4 Part (c)

Write a *single SQL query* to find stations in CA which had a *smallest rainy day precipitation value* (i.e. smallest non-zero precipitation) that was **within 400** of the *largest overall precipitation value* (across all stations & all days). Return tuples of type (station_id, min_rainy_day_precip). *Note: do not hard-code the maximum daily precipitation value, or any other values*, and **do this using GROUP BY and aggregate functions (e.g. COUNT, SUM, MAX, MIN)**. If you executed your query on the provided data, you should get:

```
station_id min_daily
88302      190
100504     90
127705     70
146002     90
196404     65
229402     70
293502     120
303805     120
357302     100
393905     70
471202     60
538802     74
652102     80
696402     100
782104     66
985505     90
```

2.5 Part (d)

Do the same as above, except do **not** use GROUP BY or any aggregate functions.

3 Problem 3: The Traveling SQL Server Salesman Problem

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 Silicon Valley, in order to make their monthly quota! Answer the following questions for a table of streets connecting company office buildings, having schema:

```
id         INT
direction  CHAR(1):  'F' or 'R'
```

```
A          TEXT
B          TEXT
d          INT
```

Note that for convenience all streets are included twice, as $A \rightarrow B$ (direction F) and $B \rightarrow A$ (direction R). 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...

Our salesperson has stopped at Stanford, to steal some cool new RDBMS technology from CS145-ers, and now wants to go sell it to a company *within 10 miles of Stanford* and *passing through no more than 3 distinct streets*. Write a single SQL query, not using WITH (see later on), to find all such companies. Your query should return tuples (company, distance) where distance is cumulative from Stanford. If you executed your query on the streets table provided, you should get:

```
company               distance
DooHickey Collective   7
DooHickey Corp        9
Gadget Collective     9
Gadget Corp           6
Gizmo Corp            9
Widget Industries     10
```

3.2 Part (b): A stop at the Farm

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 Stanford* (in order to pick up new RDBMS tech to sell!)
- A and B must *each individually* be *within 3 hops of Stanford*
- 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:

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

If you executed your query / queries on the provided data, you should get:

```
company_1             company_2             distance
DooHickey Collective  Gadget Corp           13
DooHickey Corp        Gadget Corp           15
Gadget Collective     Gadget Corp           15
Gadget Corp           DooHickey Collective  13
Gadget Corp           DooHickey Corp        15
Gadget Corp           Gadget Collective     15
Gadget Corp           Gizmo Corp            15
Gizmo Corp            Gadget Corp           15
```

3.3 Part (c): Ensuring acyclicity

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 our graph of companies and streets:

- It is connected
- It has no cycles of length > 3

Write a *single SQL query* which, if our graph is not a tree, **turns it into a tree** by deleting exactly *one* street (= two entries in our table!).

3.4 Part (d): The longest path

In this part, we want to find the distance of the longest path between any two companies. Note that you can assume that the previous part was accomplished i.e. that the graph of streets is a tree. 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 online 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, 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). If executed on the provided data, your query should return (GadgetWorks, ThingWorks, 49) *or (GadgetWorks, ThingWorks, 63)* depending on which street you deleted in Part(c).

3.5 Bonus Problem 1: The longest path, Pt. II

Using WITH RECURSIVE may be a little tricky syntactically, but it is quite elegant. What would alternatives look like? We already know we can't do it with non-recursive SQL. For this problem, **use SQL and Python** to find the longest path (by cumulative distance) between two companies in the streets graph, **without using a WITH clause**, again returning a single tuple of the form (A, B, distance).

Note: Be careful of trivial cycles in the graph! Especially if you write recursive functions in your python code, note that IPython handles hitting the max recursion depth pretty poorly (i.e. crashes / freezes up). You can debug in a normal terminal first if this is a concern.