Homework: Intermediate SQL Practice

Signal Data Science

The following problems are adapted from UC Berkeley curricular material.[1]

SQL is very important for interview questions. Consistent practice will help you retain the various details of SQL queries. Admittedly, writing SQL queries is not nearly as interesting as data science proper. However, it's sufficiently easy to master the basics that it's worth practicing nevertheless.

Refer to the SQLite documentation if you have questions about how to structure the syntax of your queries. As you work through these problems, **write down your final SQL query for each question in a separate document**.

Getting started with SQLite

In this assignment, we will be using SQLite to interactively solve some SQL problems.

- Download or install SQLite on your computer following these instructions:
  - On **Windows**, go to the SQLite download website and download the file named `sqlite-tools-win32-x86-*.zip`. Unzip the folder and check that it contains a file called `sqlite3.exe`.
  - On **OS X 10.10+**, SQLite comes pre-installed. On **OS X 10.9 or older**, go to the SQLite download website and download the file named `sqlite-tools-osx-x86-*.zip`. Unzip the folder and check that it contains a file called `sqlite3`.
  - On **GNU/Linux**, install SQLite directly from your distribution's repository, *e.g.*, `sudo apt-get install sqlite3` on Debian and Debian derivative or `sudo pacman -S sqlite3` on Arch Linux.
- Open the terminal (Command Prompt on Windows and Terminal). If SQLite was pre-installed on your system or if you installed it from a repository, you should be able to run SQLite from any directory. Type `sqlite3 --version` to verify this. If you downloaded a `.zip` file from the SQLite

[1] Specifically from CS 61A: The Structure and Interpretation of Computer Programs.

website, navigate in the terminal to the folder where `sqlite3.exe` (Windows) or `sqlite3` (OS X) is located. Type `sqlite3.exe --version` (Windows) or `./sqlite3 --version` (OS X) to verify that your downloaded binary works.

In the following, you'll be calling SQLite from the terminal. Instead of providing the exact syntax required for every operating system, we'll only provide the GNU/Linux syntax, where simply one types `sqlite3 <...>` into the terminal. Following the above instructions, adapt the syntax to your own OS. Note that if you *downloaded* the binary from the SQLite website, your terminal will have to be located in the directory as the SQLite binary to be able to call it directly; you can stay in the same directory where you unzipped the file or move the binary to a different location.

We'll illustrate how SQLite works with a simple example.

- Run `sqlite3`. You should be met with a different command prompt, saying `sqlite>`. Type `CREATE TABLE TableName as SELECT 1 AS X, 2 AS Y, 3 AS Z;` and press `Enter`. Next, type `SELECT * FROM TableName` and press `Enter`. You should receive `1|2|3` as output.

In the following exercises, we'll provide you with data for a couple simple tables. You'll design queries to answer questions about those tables, using the SQLite command line to verify that you're getting the desired output and also to allow you to interactively see how changing the structure of a query affects the output.

Company data

We've provided for your usage some `.sql` files in the `sql-homework` dataset folder. The `.sql` files will be run by SQLite directly to create the associated tables; by viewing the `.sql` files, you can directly see the data in each table.

We'll be working with a table `records` which stores informations about employees at a company, with each row corresponding to a single employee, as well as a table `meetings` which records divisional meetings. The tables are small enough that you should manually verify that your queries give the correct output by directly examining the tables' data.

- Initialize SQLite with `company.sql` by running `sqlite3 --init company.sql`. Run `SELECT * FROM records;` to view the data in the table.
- Write a query that outputs the names of employees that Oliver Warbucks directly supervises.
- Write a query that outputs all information about self-supervising employees.

- Write a query that outputs the names of all employees with salary greater than 50000 in alphabetical order.
- Write a query that outputs a table with columns: employee, salary, supervisor and supervisor's salary, containing all supervisors who earn more than twice as much as the employee.
- Write a query that outputs the names of employees whose supervisor is in a different division.
- Write a query that outputs the meeting days and times of all employees directly supervised by Oliver Warbucks.
- A middle manager is a person who is both supervising someone and is supervised by someone different. Write a query that outputs the names of all middle managers.
- Write a query that results in the names of all employees that have a meeting on the same day as their supervisor.

In the following questions, you may find the `MAX()`, `MIN()`, `SUM()`, and `COUNT(*)` functions useful.

- Write a query that outputs each supervisor and the sum of salaries of all of each supervisor's employees.
- Write a query that outputs all salaries that appear more than once in the employee records.

Recursive `WITH` statements

The `SELECT` statement can optionally include a `WITH` clause that generates and names local tables used in computing the final result. These local tables cannot be used outside of the `SELECT` statement, and they can be thought of as "helper" tables. The full syntax of a `SELECT` statement has the following form:

```
WITH [local-tables] SELECT [columns] FROM [tables]
  WHERE [condition] ORDER BY [criteria]
```

For illustration, let's consider the `company.sql` data in the previous section. For example, you can use a `WITH` statement to create and immediately use a new table to compute the final result.

```
sqlite> WITH
   ...>   schedule(day, dresscode) AS (
   ...>     SELECT "Monday",    "Sports"    UNION
   ...>     SELECT "Tuesday",   "Drag"      UNION
   ...>     SELECT "Wednesday", "Regular"   UNION
   ...>     SELECT "Thursday",  "Throwback" UNION
```

```
   ...>     SELECT "Friday",    "Casual"
   ...>   )
   ...> SELECT a.name, b.dresscode FROM
   ...>   records AS a, schedule AS b, meetings AS c
   ...>   WHERE a.division = c.division AND
   ...>   b.day = c.day ORDER BY a.name;
```

The output of the above query will be:

```
Alyssa P Hacker|Regular
Ben Bitddiddle|Regular
Cy D Fect|Regular
DeWitt Aull|Sports
DeWitt Aull|Throwback
Eben Scrooge|Sports
Lem E Tweakit|Regular
Louis Reasoner|Regular
Oliver Warbucks|Sports
Oliver Warbucks|Throwback
Robert Cratchet|Sports
```

If you try selecting from `schedule` outside, like

```
sqlite> SELECT * FROM schedule;
Error: no such table: schedule
```

you'll find that you cannot reference it because schedule was not made globally.

We can create recursive queries using the `WITH` syntax. Let's print out the natural numbers from 0 to 5 (inclusive).

We start by defining a local table called `num` that has 1 column `n`. Each row will have 1 value, a natural number. The base case is 0 (the smallest natural number) and we can create a one row table for it with `SELECT 0`. For the recursive case, for each natural number, we can add 1 to get another natural number with `SELECT n + 1` from `num`. Since we want numbers up to 5, the recursive case applies only to numbers less than 5 (so `n + 1` will be 5 or less). Finally, we combine the two cases into a single table with `UNION`.

```
sqlite> WITH num(n) AS (
   ...>   SELECT 0 UNION
   ...>   SELECT n + 1 FROM num WHERE n < 5
   ...> )
   ...> SELECT * FROM num;
```

Remember that `num` is local to this query.

- Input the above query into the SQLite command line and view the results.

Now, your turn:

- Write a query that prints out the factorial of the numbers 0 to 10 (inclusive). To do this, create a local table with two columns: the first column is the numbers 0 to 10 (inclusive), and the second column is their factorials.
- Write a query that groups natural numbers in 3-number-long sequences. Your solution should generate the following output:

```
0|1|2
3|4|5
6|7|8
9|10|11
12|13|14
```

Presidential dogs

We'll next be working with `dogs.sql`, containing three tables with information about 8 dogs, their genealogy, and their characteristics. Again, the tables here are sufficiently small that you should verify the correctness of your queries via visual inspection of the data.

The Fédération Cynologique Internationale classifies a standard poodle as over 45 cm and up to 60 cm. The `sizes` table describes this and other such classifications, where a dog must be over the `min` and less than or equal to the `max` in height to qualify as a `size`.

- Output a table that has a column of the names of all dogs that have a `parent`, ordered by the height of the parent from tallest parent to shortest parent. For example, `fillmore` has a parent (`eisenhower`) with height 35, and so should appear before `grover` who has a parent (`fillmore`) with height 32. The names of dogs with parents of the same height should appear together in any order. For example, `barack` and `clinton` should both appear at the end, but either one can come before the other.
- Create a single string for every pair of siblings that have the same size. Each value should be a sentence describing the siblings by their size, and each sibling pair should appear only once in alphabetical order.

  The expected output is:

```
barack and clinton are standard siblings
abraham and grover are toy siblings
```

  Hints: (1) First use a `WITH` clause to create a local table of siblings. Comparing the size of siblings will be simplified. (2) If you join a table with itself, use `AS` within the `FROM` clause to give each table an alias. (3) In order to concatenate two strings into one, use the `||` operator.

When dogs are stacked on top of one another, the total height of the stack is the sum of the heights of the dogs.

- Create a two-column table describing all stacks of 4 dogs at least 170 cm high. The first column should contains a comma-separated list of dogs in the stack, and the second column should contain the total height of the stack. Order the stacks in increasing order of total height. A valid stack of dogs includes each dog only once, and the dogs should be listed in increasing order of height within the stack. **Assume that no two dogs have the same height.**

  The expected output is:

  ```
  abraham, delano, clinton, barack|171
  grover, delano, clinton, barack|173
  herbert, delano, clinton, barack|176
  fillmore, delano, clinton, barack|177
  eisenhower, delano, clinton, barack|180
  ```

  Hints: (1) Use a `WITH` clause to create a recursive table with additional columns, such as the number of dogs that have been stacked and information about the last dog added (to control the dog order). Then, select the rows and columns from this larger table to generate the final solution. (2) Use height comparisons to ensure that dogs are not repeated in a stack. (3) Generating the comma-separated list of dogs is easier if your base case includes the name of one dog without any commas before or after it, rather than no dogs at all.

- Write a query which outputs a table that includes the `height` and `name` of every dog that shares the 10s digit of its `height` with at least one other dog and has the highest 1's digit of all dogs that have the same 10's digit.

A non-parent relation is either an ancestor that is not a parent (such as a grandparent or great-grandparent) or a descendent that is not a child (such as a grandchild or great-grandchild). Siblings are *not* relations under this definition.

- Select all pairs that form non-parent relations ordered by the difference in height between one dog and the other. The shortest paired with the tallest should appear first, and the tallest paired with the shortest should appear last. If two pairs have the same height difference, they may appear together in any order.