Instructions

Please download lab materials `lab11.zip` from our QQ group if you don't have one.

In this lab, you are required to complete the problems described in section 4. The starter code for these problems is provided in `lab11.sql`.

Submission: As instructed before, you need to submit your work with Ok by `python ok --submit`. You may submit more than once before the deadline, and your score of this assignment will be the highest one of all your submissions.

Readings: You might find the following references to the textbook useful:

- Section 4.3 - Declarative Programming

Usage

First, check that a file named `sqlite_shell.py` exists alongside the assignment files. If you don't see it, or if you encounter problems with it, read the Troubleshooting section to see how to download an official precompiled SQLite binary before proceeding.

You can start an interactive SQLite session in your Terminal with the following command:

```
$ cd to/the/code/directory
$ python sqlite_shell.py
```

Following prompts are expected:

```
SQLite version 3.35.5 (adapter version 2.6.0)
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
sqlite>
```

While the interpreter is running, you can type `.help` to see some of the commands you can run.

To exit out of the SQLite interpreter, type `.exit` or `.quit` or press `Ctrl-C`. Remember that if you see `...>` after pressing enter, you probably forgot a `;`.

You can also run all the statements in a `.sql` file by doing the following:

1. Runs your code and then opens an interactive SQLite session, which is similar to running Python code with the interactive `-i` flag.

   ```
   python sqlite_shell.py --init lab11.sql
   ```

2. (Linux, macOS) Runs your code and then exits SQLite immediately afterwards.

   ```
   python sqlite_shell.py < lab11.sql
   ```

Troubleshooting

> If you do not have problems using the `sqlite_shell.py` described before, you do not need to read this section.

Python already comes with a built-in SQLite database engine to process SQL. However, it doesn't come with a "shell" to let you interact with it from the terminal. Because of this, until now, you have been using a simplified SQLite shell written by us. However, you may find the shell is old, buggy, or lacking in features. In that case, you may want to download and use the official SQLite executable.

If running `python3 sqlite_shell.py` didn't work, you can download a precompiled sqlite directly by following the following instructions and then use `sqlite3` and `./sqlite3` instead of `python3 sqlite_shell.py` based on which is specified for your platform.

Another way to start using SQLite is to download a precompiled binary from the SQLite website.

SQLite version 3.32.3 or higher should be sufficient.

However, before proceeding, please remove (or rename) any SQLite executables (`sqlite3`, `sqlite_shell.py`, and the like) from the current folder, or they may conflict with the official one you download below. Similarly, if you wish to switch back later, please remove or rename the one you downloaded and restore the files you removed.

Windows

1. Visit the download page linked above and navigate to the section Precompiled Binaries for Windows. Click on the link **sqlite-tools-win32-x86-*.zip** to download the binary.
2. Unzip the file. There should be a `sqlite3.exe` file in the directory after extraction.
3. Navigate to the folder containing the `sqlite3.exe` file and check that the version is at least 3.32.3:

```
$ cd path/to/sqlite
$ ./sqlite3 --version
3.32.3 2020-06-18 14:16:19
```

macOS Big Sur (11.0.1) or newer

SQLite comes pre-installed. Check that you have a version that's greater than 3.32.3:

```
$ sqlite3
SQLite version 3.32.3
```

macOS (older versions)

SQLite comes pre-installed, but it may be the wrong version. You can take the following steps if the pre-installed version is less than 3.32.3.

1. Visit the download page linked above and navigate to the section **Precompiled Binaries for Mac OS X (x86)**. Click on the link **sqlite-tools-osx-x86-*.zip** to download the binary.
2. Unzip the file. There should be a `sqlite3` file in the directory after extraction.
3. Navigate to the folder containing the `sqlite3` file and check that the version is at least 3.32.3:

```
$ cd path/to/sqlite
$ ./sqlite3 --version
3.32.3 2020-06-18 14:16:19
```

Ubuntu

The easiest way to use SQLite on Ubuntu is to install it straight from the native repositories (the version will be slightly behind the most recent release). Check that the version is greater than 3.32.3:

```
$ sudo apt install sqlite3
$ sqlite3 --version
3.32.3 2020-06-18 14:16:19
```

Review

Consult this section if you need a refresher on the material for this lab. It's okay to skip directly to the next section and refer back here when you get stuck.

SQL Aggregation

Previously, we have been dealing with queries that process one row at a time. When we join, we make pairwise combinations of all of the rows. When we use `WHERE`, we filter out certain rows based on the condition. Alternatively, applying an aggregate function such as `MAX(column)` combines the values in multiple rows.

```
CREATE TABLE flights AS
  SELECT "AUH" AS departure, 932 AS price UNION
  SELECT "LAS",              50           UNION
  SELECT "LAX",              89           UNION
  SELECT "LAX",              99           UNION
  SELECT "SEA",              32           UNION
  SELECT "SFO",              50           UNION
  SELECT "SFO",              40           UNION
  SELECT "SFO",              60           UNION
  SELECT "SLC",              49           UNION
  SELECT "SLC",              42;
```

By default, we combine the values of the *entire* table. For example, if we wanted to count the number of flights from our `flights` table, we could use:

```
sqlite> SELECT COUNT(*) from flights;
10
```

GROUP BY

What if we wanted to group together the values in similar rows and perform the aggregation operations within those groups? We use a `GROUP BY` clause.

Here's another example. For each unique departure, collect all the rows having the same departure airport into a group. Then, select the `price` column and apply the `MIN` aggregation to recover the price of the cheapest departure from that group. The end result is a table of departure airports and the cheapest departing flight.

```
sqlite> SELECT departure, MIN(price) FROM flights GROUP BY departure;
AUH|932
LAS|50
LAX|89
SEA|32
SFO|40
SLC|42
```

HAVING

Just like how we can filter out rows with `WHERE`, we can also filter out groups with `HAVING`. Typically, a `HAVING` clause should use an aggregation function. Suppose we want to see all airports with at least two departures:

```
sqlite> SELECT departure FROM flights GROUP BY departure HAVING COUNT(*) >= 2;
LAX
SFO
SLC
```

DISTINCT

Note that the `COUNT(*)` aggregate just counts the number of rows in each group. Say we want to count the number of *distinct* airports instead. Then, we could use the following query:

```
sqlite> SELECT COUNT(DISTINCT departure) FROM flights;
6
```

This enumerates all the different departure airports available in our `flights` table (in this case: SFO, LAX, AUH, SLC, SEA, and LAS).

Required Problems

In this section, you are required to complete the problems below and submit your code to OJ website.

Remember, you can use `ok` to test your code:

```
$ python ok  # test all tables
$ python ok -q <table>  # test single table
```

You can also do your own experiments interactively:

```
$ python sqlite_shell.py --init lab11.sql
```

Or only starts with the original data in `lab11_data.sql`:

```
$ python sqlite_shell.py --init lab11_data.sql
```

Survey Data

Long long ago, *cs61a* asked their students to complete a brief online survey through Google Forms, which involved relatively random but fun questions. In this lab, we will interact with the results of the survey by using SQL queries to see if we can find interesting things in the data.

First, take a look at `lab11_data.sql` and examine the table defined in it. Note its structure. You will be working with the two tables (`students` and `numbers`) in this file.

students

The first is the table `students`, which is the main results of the survey. Each column represents a different question from the survey, except for the first column, which is the time of when the result was submitted. The time is a *unique* identifier for each of the rows in the table. The last several columns all correspond to the last question on the survey (more details below.)

Column Name	Question
`time`	The unique timestamp that identifies the submission
`number`	What's your favorite number between 1 and 100?
`color`	What is your favorite color?
`seven`	Choose the number 7 below. Options: • 7 • Choose this option instead • seven • the number 7 below. • You're not the boss of me!
`song`	If you could listen to only one of these songs for the rest of your life, which would it be? Options: • "Smells Like Teen Spirit" by Nirvana • "Shelter" by Porter Robinson • "Clair de Lune" by Claude Debussy • "Dancing Queen" by ABBA • "Down With The Sickness" by Disturbed • "Everytime We Touch" by Cascada • "All I want for Christmas is you" by Mariah Carey • "STAY" by The Kid LAROI, Justin Bieber • "Old Town Road" by Lil Nas X • "Turntables" by Janelle Monáe • "Shake It Off" by Taylor Swift
`date`	Pick a day of the year!
`pet`	If you could have any animal in the world as a pet, what would it be?
`instructor`	Choose your favorite photo of John DeNero
`smallest`	Try to guess the smallest unique positive INTEGER that anyone will put!

numbers

The second table is `numbers`, which is the results from the survey in which students could select more than one option from the numbers listed, which ranged from 0 to 10 and included 2021, 2022, 9000, and 9001. Each row has a time (which is again a unique identifier) and has the value `"True"` if the student selected the column or `"False"` if the student did not. The column names in this table are the following strings, referring to each possible number: `0`, `1`, `2`, `4`, `5`, `6`, `7`, `8`, `9`, `10`, `2021`, `2022`, `9000`, `9001`.

Since the survey was anonymous, we used the timestamp that a survey was submitted as a unique identifier. A time in `students` matches up with a time in `numbers`. For example, a row in `students` whose `time` value is `"11/17/2021 10:52:40"` matches up with the row in `numbers` whose `time` value is `"11/17/2021 10:52:40"`. These entries come from the same Google form submission and thus belong to the same student.

Problem 1: Go Bears! (And Dogs?) (100 pts)

Now that we have learned how to select columns from a SQL table, let's filter the results to see some more interesting results!

It turns out that 61A students have a lot of school spirit: the most popular favorite color was `"blue"`. You would think that this school spirit would carry over to the pet answer, and everyone would want a pet bear! Unfortunately, this was not the case, and the majority of students opted to have a pet `"dog"` instead. That is the more sensible choice, I suppose...

bluedog

Write a SQL query to create a table that contains both the column `color` and the column `pet`, using the keyword `WHERE` to restrict the answers to the most popular results of color being `"blue"` and pet being `"dog"`.

You should get the following output:

```
sqlite> SELECT * FROM bluedog;
blue|dog
blue|dog
blue|dog
blue|dog
```

```
CREATE TABLE bluedog AS
  SELECT "REPLACE THIS LINE WITH YOUR SOLUTION";
```

bluedog_songs

This isn't a very exciting table, though. Each of these rows represents a different student, but all this table can really tell us is how many students both like the color blue and want a dog as a pet, because we didn't select for any other identifying characteristics. Let's create another table, `bluedog_songs`, that looks just like `bluedog` but also tells us how each student answered the `song` question.

You should get the following output:

```
sqlite> SELECT * FROM bluedog_songs;
blue|dog|Clair De Lune
blue|dog|Shake It Off
blue|dog|Old Town Road
blue|dog|Dancing Queen
```

```
CREATE TABLE bluedog_songs AS
  SELECT "REPLACE THIS LINE WITH YOUR SOLUTION";
```

This distribution of songs actually largely represents the distribution of song choices that the total group of students made, so perhaps all we've learned here is that there isn't a correlation between a student's favorite color and desired pet, and what song they could spend the rest of their life listening to. Even demonstrating that there is no correlation still reveals facts about our data though!

Problem 2: The Smallest Unique Positive Integer (100 pts)

Who successfully managed to guess the smallest unique positive integer value? Let's find out!

Write an SQL query to create a table with the columns `time` and `smallest` which contains the timestamp for each submission that made a unique guess for the smallest unique positive integer - that is, only one person put that number for their guess of the smallest unique integer. Also include their guess in the output.

> *Hint:* Think about what attribute you need to `GROUP BY`. Which groups do we want to keep after this? We can filter this out using a `HAVING` clause. If you need a refresher on aggregation, see the Review section.

The submission with the timestamp corresponding to the minimum value of this table is the timestamp of the submission with the smallest unique positive integer!

```
CREATE TABLE smallest_int_having AS
  SELECT "REPLACE THIS LINE WITH YOUR SOLUTION";
```

Problem 3: Matchmaker, Matchmaker (100 pts)

Did you take 61A with the hope of finding a new group of friends? Well you're in luck! With all this data in hand, it's easy for us to find your perfect match. If two students want the same pet and have the same taste in music, they are clearly meant to be friends! In order to provide some more information for the potential pair to converse about, let's include the favorite colors of the two individuals as well!

In order to match up students, you will have to do a *join* on the `students` table with itself. When you do a join, SQLite will match every single row with every single other row, so make sure you do not match anyone with themselves, or match any given pair twice!

> **Important Note:** When pairing the first and second person, make sure that the first person responded first (i.e., the first person has an earlier `time`). This is to ensure your output matches our tests.

Write a SQL query to create a table that has 4 columns:

- The shared preferred `pet` of the pair
- The shared favorite `song` of the pair
- The favorite `color` of the first person
- The favorite `color` of the second person

```
CREATE TABLE matchmaker AS
  SELECT "REPLACE THIS LINE WITH YOUR SOLUTION";
```

> *Hint:* When joining table names where column names are the same, use dot notation to distinguish which columns are from which table: `[table_name].[column name]`. This sometimes may get verbose, so it's stylistically better to give tables an alias using the `AS` keyword. The syntax for this is as follows:
>
> ```
> SELECT <[alias1].[column_name1], [alias2].[column_name2]...>
>     FROM <[table_name1] AS [alias1],[table_name2] AS [alias2]...> ...
> ```

Problem 4: Sevens (100 pts)

Let's take a look at data from both of our tables, `students` and `numbers`, to find out if students that really like the number 7 also chose `'7'` for the obedience question. Specifically, we want to look at the students that fulfill the below conditions:

Conditions:

- reported that their favorite number (column `number` in `students`) was 7
- have `"True"` in column `"7"` in `numbers`, meaning they checked the number 7 during the survey

In order to examine rows from both the `students` and the `numbers` table, we will need to perform a *join*.

How would you specify the `WHERE` clause to make the `SELECT` statement only consider rows in the joined table whose values all correspond to the same student? If you find that your output is massive and overwhelming, then you are probably missing the necessary condition in your `WHERE` clause to ensure this.

> *Note:* The columns in the `numbers` table are strings with the associated number, so you must put quotes around the column name to refer to it. For example if you alias the table as `a`, to get the column to see if a student checked 9001, you must write `a."9001"`.

Write a SQL query to create a table with just the column `seven` from `students`, filtering first for students who said their favorite number (column `number`) was 7 in the `students` table and who checked the box for seven (column `"7"`) in the `numbers` table.

```
CREATE TABLE sevens AS
  SELECT "REPLACE THIS LINE WITH YOUR SOLUTION";
```

The first 10 lines of this table should be:

```
sqlite> SELECT * FROM sevens LIMIT 10;
seven
7
7
7
7
```

Problem 5: Average Difference (100 pts)

Let's try to find the average absolute difference between every person's favorite number `number` and their guess at the smallest number anyone will put, `smallest`, rounded to the closest number.

For example, suppose two students put the following:

```
| number | smallest |
| ------ | -------- |
| 10     | 1        |
| 2      | 3        |
```

Then, average absolute difference is (abs(10-1) + abs(2-3)) / 2 = 5.0.

Create a table that contains one row with one value, the rounded value of the average absolute difference between every person's favorite number and their guess at the smallest value.

Hints:

- `abs` is a `sqlite3` function that returns the absolute value of the argument.
- `round` is a function that rounds the value of the argument.
- `avg` is a function that returns the average value in a set.

```
CREATE TABLE avg_difference AS
  SELECT "REPLACE THIS LINE WITH YOUR SOLUTION";
```