Lab 09: SQL

1. Instructions

> Please download lab materials `lab09.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 `lab09.sql`, which are distributed as part of the lab materials in the `code` directory. You should also take a look at the `lab09data.sql` that contains data of answers to a questionnaire finished by CS61A students before (Hope that TAs of SICP2021 will collect our own data, Hah). A detailed description can be found in section 4. You only have to make changes to `lab09.sql` in this lab.

Submission: When you are done, submit your code to our Grader server `python submit.py --stuid <YOUR STUDENT ID> --stuname <YOUR NAME>`. You may submit more than once before the deadline; grader will record the highest score graded from your submissions. Check that you have successfully submitted your code and what your score is on Grader website.

> See lab07 for more instructions on on submitting assignments.

> **WARNING:** Do not modify `submit.py`!

Using Ok: If you have any questions about using Ok, please refer to this guide.

Readings: You might find the following references useful:

- Section 4.3 - Declarative Programming

2. 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, scroll down to 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
```

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 exits SQLite immediately afterwards.

```
python sqlite_shell.py < lab09.sql
```

2. 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 lab09.sql
```

To check your progress, you can run `sqlite3` directly by running:

```
python sqlite_shell.py --init lab09.sql
```

You should also check your work using ok:

```
python ok --local
```

3. Review

Consult this section if you need a refresher on the material for this lab, or if you're having trouble running SQL or SQLite on your computer. It's okay to skip directly to the questions and refer back here should you get stuck.

3.1 SQL

SQL is a declarative programming language. Unlike Python or Scheme where we write programs which provide the exact sequence of steps needed to solve a problem, SQL accepts instructions which express the desired result of the computation.

The challenge with writing SQL statements then is in determining how to compose the desired result! SQL has a strict syntax and a structured method of computation, so even though we write statements which express the desired result, we must still keep in mind the steps that SQL will follow to compute the result.

SQL operates on *tables* of data, which contains a number of fixed *columns*. Each row of a table represents an individual data point, with values for each column. SQL statements then operate on these tables by iterating over each row, determining if it should be included in the output relation (filtering), and then computing the resulting value which should appear in the table.

We can also describe SQL's implementation using the following code as an example. Imagine the `SELECT`, `FROM`, `WHERE`, and `ORDER BY` clauses are implemented as functions which act on rows. Here's a simplified view of how SQL might work, if implemented in simple Python.

```
output_table = []
for row in FROM(*input_tables):
    if WHERE(row):
        output_table += [SELECT(row)]
if ORDER_BY:
    output_table = ORDER_BY(output_table)
if LIMIT:
    output_table = output_table[:LIMIT]
```

Note that the `ORDER BY` and `LIMIT` clauses are applied only at the end after all the rows in the output table have been determined.

One of the important things to remember about SQL is that we always return to this very simple model of computation: looping, filtering, applying a function, and then ordering and limiting the final output.

The simple Python example above helps expose a limitation of SQL: we currently can't create output tables with more rows than in the input! There are a few methods for creating novel combinations of existing data: **joins** and SQL **recursion**. **Aggregation** allows us to find patterns and consider multiple rows together as a single unit, or group.

3.2 Joins

Joins create novel combinations of data by combining data from more than one source. Given multiple input tables, we can combine them in a join. Following the Python metaphor, the join is like creating nested `for` loops.

```
def FROM(table_1, table_2):
    for row_1 in table1:
        for row_2 in table2:
            yield row_1 + row_2
```

Given each row in `table_1` and each row in `table_2`, the join iterates over each possible combination of rows and treats them as the input table. The same idea extends to more than two tables as well.

Joins are particularly useful when we want to combine data on a single column. For example, say we have a table, `dogs`, containing the `name` and `size` of each dog, and a different table, `parents`, containing the `name` and `parent` of each dog. We might want to ask, "What's the difference in size between each dog and their parent?" by joining together the tables in a SQL statement.

The first question we should ask ourselves is, "Which data tables do we need to reference to assemble all the data we need?" We'll definitely need the table of `parents` to determine the name of each dog and their parent. From their names, we still need a way to get the size of each dog. That information is provided by the `dogs` table.

```
SELECT d.name, d.size, p.parent
  FROM dogs as d, parents as p
 WHERE d.name = p.name;
```

But referencing the `dogs` table only once will leave us in a tricky situation. We can find either the size of the dog or their parent, but not both!

```
SELECT d1.name, d1.size, d2.name, d2.size
  FROM dogs as d1, dogs as d2, parents as p
 WHERE d1.name = p.name AND p.parent = d2.name;
```

Joining the `dogs` table twice provides the necessary information to solve the problem.

3.3 Aggregation

We saw joins as a method for creating novel combinations of data, and recursion as an extension of joins. These methods combine data by extending the number of columns we have available to us and help us identify the patterns in data.

Aggregation functions allow us to operate on data in a different way by combining results across multiple rows. Common aggregation functions to be familiar with include `COUNT`, `MIN`, `MAX`, `SUM`, and `AVG`.

Applying an aggregation function to an input relation results in a single row containing the aggregate result.

```
> SELECT AVG(n) FROM n5;
3.0
```

But oftentimes, we'd like to condition the groups and compute aggregate results for smaller portions of the input relation. We can use `GROUP BY` and `HAVING` to split the rows into groups and select only a subset of the groups.

```
output_table = []
for input_group in GROUP_BY(FROM(*input_tables)):
    output_group = []
    for row in input_group:
        if WHERE(row):
            output_group += [row]
    if HAVING(output_group):
        output_table += [SELECT(output_group)]
if ORDER_BY:
    output_table = ORDER_BY(output_table)
if LIMIT:
    output_table = output_table[:LIMIT]
```

We take the results from the input tables, whether it's just a single table or a join, and then apply the same row-by-row processing *within* a group. Before adding the result of the group to the output table, we check to see if the values of the group reflect the condition in the `HAVING` clause which serves as a filter on the groups, much like how `WHERE` is a filter on the rows.

Once we have groups, we can aggregate over the groups in our table and find things like:

- the maximum value (`MAX`),
- the minimum value (`MIN`),
- the number of rows in the group (`COUNT`),
- the average over all of the values (`AVG`),

3.3.1 The COUNT Aggregator

`COUNT` will count the number of rows in each group. For example, the following query will print out the top 10 favorite numbers with their respective counts:

```
sqlite> SELECT number, COUNT(*) AS count FROM students GROUP BY number ORDER BY
count DESC LIMIT 10;
7|7
2|5
6|5
12|5
21|5
27|5
69|5
99|5
13|4
19|4
```

This `SELECT` statement first groups all of the rows in our table students by number. Then, within each group, we perform aggregation by COUNTing all the rows. By selecting number and `COUNT(*)`, we then can see the highest number and how many students picked that number. We have to order by our `COUNT(*)`, which is saved in the alias count, by DESCending order, so our highest count starts at the top, and we limit our result to the top 10.

3.3.2 ORDER BY

You can add `ORDER BY` column to the end of any query to sort the results by that column, in ascending order.

4. Required Problems

Data

CS61A students were asked 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. For convenience, we use their data directly.

First, take a look at `lab09data.sql` and examine the table defined in it. Note its structure. You will be working with:

- `students`: 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. This time is a unique identifier for each of the rows in the table.

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 - You're not the boss of me! - Choose this option instead - seven - the number 7 below.
`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 - "The Middle" by Zedd - "Clair de Lune" by Claude Debussy - "Finesse ft. Cardi B" by Bruno Mars - "Down With The Sickness" by Disturbed - "Everytime We Touch" by Cascada - "All I want for Christmas is you" by Mariah Carey - "thank u, next" by Ariana Grande
`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!

- `checkboxes`: 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 2019, 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'`, `'2019'`, `'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 `checkboxes`. For example, a row in `students` whose `time` value is `"2019/08/06 4:19:18 PM MDT"` matches up with the row in `checkboxes` whose `time` value is `"2019/08/06 4:19:18 PM MDT"`. These entries come from the same Google form submission and thus belong to the same student.

You will write all of your solutions in the starter file `lab09.sql` provided. As with other labs, you can test your solutions with OK. In addition, you can use either of the following commands:

```
python sqlite_shell.py < lab09.sql
python sqlite_shell.py --init lab09.sql
```

Problem 1: What Would SQL Print (0 pts)

> **Note**: there is no submission for this question

First, load the tables into sqlite3.

```
$ python sqlite_shell.py --init lab09.sql
```

Before we start, inspect the schema of the tables that we've created for you:

```
sqlite> .schema
```

This tells you the name of each of our tables and their attributes.

Let's also take a look at some of the entries in our table. There are a lot of entries though, so let's just output the first 20:

```
sqlite> SELECT * FROM students LIMIT 20;
```

If you're curious about some of the answers of CS61A students, open up `lab09data.sql` in your favorite text editor and take a look (We may collect our own data in future semesters)!

For each of the SQL queries below, think about what the query is looking for, then try running the query yourself and see!

```
sqlite> SELECT * FROM students LIMIT 30; -- This is a comment. * is shorthand
for all columns!
_______

sqlite> SELECT color FROM students WHERE number = 7;
_______

sqlite> SELECT song, pet FROM students WHERE color = "blue" AND date = "12/25";
_______
```

Remember to end each statement with a `;`! To exit out of SQLite, type `.exit` or `.quit` or hit `Ctrl-C`.

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

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

While we could find out the smallest unique integer using aggregation, for now let's just try hand-inspecting the data. An anonymous elf has informed us that the smallest unique positive value is greater than 2.

Write an SQL query to create a table with the columns `time` and `smallest` that we can inspect to determine what the smallest integer value is. In order to make it easier for us to inspect these values, use `WHERE` to restrict the answers to numbers greater than 2, `ORDER BY` to sort the numerical values, and `LIMIT` your result to the first 20 values that are greater than the number 2.

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

Use Ok to test your code:

```
python ok -q smallest-int --local
```

After you've successfully passed the Ok test, take a look at the table `smallest_int` that you just created and manually find the smallest unique integer value!

To do this, try the following:

```
$ python sqlite_shell.py --init lab09.sql
sqlite> SELECT * FROM smallest_int; -- No LIMIT this time!
```

Problem 3: Matchmaker, Matchmaker (100 pts)

Did you take SICP with the hope of finding your soul mate? Well you're in luck (Sadly, you can only conduct matchmaking for the CS61A students this semester. :( )! With all the data in hand, it's easy for us to find their perfect match. If two students want the same pet and have the same taste in music, they are clearly meant to be together! In order to provide some more information for the

potential lovebirds 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. they have an earlier `time`). This is to ensure your output matches our tests.
>
> *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].[columnname2]...>
> FROM <[table_name1] AS [alias1],[table_name2] AS [alias2]...> ...
> ```
>
> The query in the football example from earlier uses this syntax.

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

- The shared preferred `pet` of the couple
- The shared favorite `song` of the couple
- 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";
```

Use Ok to test your code:

```
python ok --local -q matchmaker
```

Problem 4: 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 topics 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";
```

Use Ok to test your code:

```
python ok --local -q smallest-int-having
```

Problem 5: Let's Count (100 pts)

Let's have some fun with this! For each query below, we created its own table in `lab09.sql`, so fill in the corresponding table and run it using Ok. Try working on this on your own or with a neighbor before toggling to see the solutions.

> *Hint:* You may find that there isn't a particular attribute you should have to perform the `COUNT` aggregation over. If you are only interested in counting the number of rows in a group, you can just say `COUNT(*)`.

What are the top 10 pets this semester?

```
sqlite> SELECT * FROM sicp20favpets;
  dog|15
  cat|9
  lion|4
  cheetah|3
  golden retriever|3
  pig|3
  corgi|2
  horse|2
  human|2
  koala|2
```

```
CREATE TABLE sicp20favpets AS
  SELECT "REPLACE THIS LINE WITH YOUR SOLUTION";
```

How many people marked exactly the word 'dog' as their ideal pet this semester?

```
sqlite> SELECT * FROM sicp20dog;
  dog|15
```

```
CREATE TABLE sicp20dog AS
  SELECT "REPLACE THIS LINE WITH YOUR SOLUTION";
```

The possibilities are endless, so have fun experimenting!

Use Ok to test your code:

```
python ok -q lets-count --local
```

5. Troubleshooting

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 provided by CS61A. However, you may find the shell is old, buggy, or lacking in features (These comments are from CS61A themselves :)). In that case, you may want to download and use the official SQLite executable based on the following instructions.

Alternatives to SQLite shell

If running `python 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 `python 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. The latest version of SQLite at the time of writing is 3.28.0, but you can check for additional updates on the website.
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 3.8.3:

   ```
   $ cd path/to/sqlite
   $ ./sqlite3 --version
   3.12.1 2016-04-08 15:09:49 fe7d3b75fe1bde41511b323925af8ae1b910bc4d
   ```

macOS Yosemite (10.10) or newer

SQLite comes pre-installed. Check that you have a version that's greater than 3.8.3:

```
$ sqlite3
SQLite version 3.8.10.2
```

Mac OS X Mavericks (10.9) or older

SQLite comes pre-installed, but it is the wrong version.

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 3.8.3:

```
$ cd path/to/sqlite
$ ./sqlite3 --version
3.12.1 2016-04-08 15:09:49 fe7d3b75fe1bde41511b323925af8ae1b910bc4d
```

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):

```
$ sudo apt install sqlite3
$ sqlite3 --version
3.8.6 2014-08-15 11:46:33 9491ba7d738528f168657adb43a198238abde19e
```