Lab 11 Solutions **lab11.zip (lab11.zip)**

Solution Files

Topics

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.

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.

The SQL lab also has a great visual demonstrating this exact result 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.n
```

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.

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.

SQL Survey Questions (Required)

Last week, we asked you and your fellow 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 `sp20data.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 (Options shown under Question 3)
`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.

> *Note*: If you are looking for your personal response within the data, you may have noticed that some of your answers are slightly different from what you had input. In order to make SQLite accept our data, and to optimize for as many matches as possible during our joins, we did the following things to clean up the data:
>
> - `color` and `pet`: We converted all the strings to be completely lowercase.
> - For some of the more "free-spirited" responses, we escaped the special characters so that they could be properly parsed.

You will write all of your solutions in the starter file `lab11.sql` provided. As with other labs, you can test your solutions with OK. In addition, you can use either of the following commands:

```
python3 sqlite_shell.py < lab11.sql
python3 sqlite_shell.py --init lab11.sql
```

Q1: What Would SQL print?

> Note: there is no submission for this question

First, load the tables into sqlite3.

```
$ python3 sqlite_shell.py --init lab11.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 students put into the Google form, open up `sp20data.sql` in your favorite text editor and take a look!

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 fo
------

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`.

Q2: Obedience

To warm-up, let's ask a simple question related to our data: Is there a correlation between whether students do as they're told and their favorite images of their instructors?


Image 1

Image 2


Image 3


Image 4

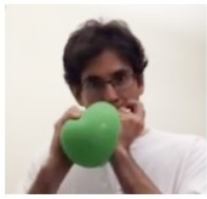
Image 5

Write an SQL query to create a table that contains the columns `seven` (this column representing "obedience") and `instructor` (the image of the instructor students selected) from the `students` table.

```
CREATE TABLE obedience AS
  SELECT seven, instructor FROM students;

  -- Video walkthrough: https://youtu.be/Og2RoZPRk_c?t=40m43s
```

Use Ok to test your code:

```
python3 ok -q obedience
```

Q3: The Smallest Unique Positive Integer

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 time, smallest FROM students WHERE smallest > 2 ORDER BY smallest LIMI

-- Video walkthrough: https://youtu.be/Og2RoZPRk_c?t=41m47s
-- Note: Minor difference, walkthrough uses smallest > 15 instead of 2.
```

Use Ok to test your code:

```
python3 ok -q smallest-int
```

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:

```
$ python3 sqlite_shell.py --init lab11.sql
sqlite> SELECT * FROM smallest_int; -- No LIMIT this time!
```

Q4: Matchmaker, Matchmaker

Did you take 61A with the hope of finding your soul mate? 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 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 a.pet, a.song, a.color, b.color FROM students AS a, students AS b
    WHERE a.time < b.time AND a.pet = b.pet AND a.song = b.song;

  -- Video walkthrough: https://youtu.be/Og2RoZPRk_c?t=1h17m40s
```

Use Ok to test your code:

```
python3 ok -q matchmaker
```

Submit

Make sure to submit this assignment by running:

```
python3 ok --submit
```

Optional Questions

NOTE: These questions require Aggregation, which is not covered in lecture until Wednesday, so please watch that lecture before starting on these problems!

Q5: The Smallest Unique Positive Integer

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 time, smallest FROM students GROUP BY smallest HAVING COUNT(*) = 1;
```

Use Ok to test your code:

```
python3 ok -q smallest-int-having
```

The COUNT Aggregator

How many people liked each pet? What is the biggest date chosen this semester? How many obedient people chose Image 1 for the instructor? Is there a difference between last semester's average favorite number and this semester's?

To answer these types of questions, we can bring in SQL aggregation, which allows us to accumulate values across rows in our SQL database!

In order to perform SQL aggregation, we can group rows in our table by one or more attributes. 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`),

and more! `SELECT` statements that use aggregation are usually marked by two things: an aggregate function (`MAX`, `MIN`, `COUNT`, `AVG`, etc.) and a `GROUP BY` clause. `GROUP BY [column(s)]` groups together rows with the same value in each column(s).

In this section we will only use `COUNT`, which will count the number of rows in each group, but feel free to check out this link (http://www.sqlcourse2.com/agg_functions.html) for more!

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 `COUNT`ing 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 `DESC` ending order, so our highest count starts at the top, and we limit our result to the top 10.

ORDER BY

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

Q6: Let's Count

Let's have some fun with this! For each query below, we created its own table in `lab11.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 sp20favpets;
dog|15
cat|9
lion|4
cheetah|3
golden retriever|3
pig|3
corgi|2
horse|2
human|2
koala|2
```

```
CREATE TABLE sp20favpets AS
  SELECT pet, COUNT(*) AS count FROM students
    GROUP BY pet ORDER BY count DESC LIMIT 10;
```

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

```
sqlite> SELECT * FROM sp20dog;
dog|15
```

```
CREATE TABLE sp20dog AS
  SELECT pet, COUNT(*) FROM students WHERE pet = 'dog';
```

We can find the student's favorite for any column (try it yourself in the interpreter), but let's go back to our Obedience question. Let's see how many obedient students this semester picked each image of an instructor. We can do this by selecting only the rows that have `seven = '7'` then `GROUP BY instructor`, and finally we can `COUNT` them.

```
sqlite> SELECT * FROM obedienceimages;
7|1|6
7|2|7
7|4|3
7|5|7
7|6|11
```

```
CREATE TABLE obedienceimages AS
  SELECT seven, instructor, COUNT(*) FROM students WHERE seven = '7'
    GROUP BY instructor;
```

The possibilities are endless, so have fun experimenting!

Use Ok to test your code:

```
python3 ok -q lets-count
```

Other Optional SQL Questions

Q7: Stacks

Sufficiently sure-footed dogs can stand on either other's backs to form a stack (up to a point). We'll say that the total height of such a stack is the sum of the heights of the dogs.
Create a two-column table describing all stacks of four dogs at least 170 cm high. The first column should contain 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.

> Note: there are no stacks of less than 4 dogs that reach 170cm in height.

```
-- Ways to stack 4 dogs to a height of at least 170, ordered by total height
CREATE TABLE stacks_helper(dogs, stack_height, last_height);

INSERT INTO stacks_helper SELECT name, height, height FROM dogs;
INSERT INTO stacks_helper SELECT dogs || ", " || name, stack_height + height, h
INSERT INTO stacks_helper SELECT dogs || ", " || name, stack_height + height, h
INSERT INTO stacks_helper SELECT dogs || ", " || name, stack_height + height, h

CREATE TABLE stacks AS
  SELECT dogs, stack_height FROM stacks_helper WHERE stack_height >= 170 ORDER
```

A valid stack of dogs includes each dog only once, and the dogs should be listed in increasing order of height within the stack. You may assume that no two dogs have the same height.

```
sqlite> select * from stacks;
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
```

You should use the provided helper table `stacks_helper`. It has 3 columns: (1) `dogs` - a stack of dogs as a comma separated list of dog names, (2) `stack_height` - the height of the stack, and (3) `last_height` - the height of the last dog added to the stack (in order to ensure we have the right order in the stack).

First, fill this table up by doing the following:

1. Use an `INSERT INTO` to add stacks of just one dog into `stacks_helper`. You can use this syntax to insert rows from a table called `t1` into a table called `t2`:

   ```
   INSERT INTO t2 SELECT [expression] FROM t1 ...;
   ```

   For example:

   ```
   sqlite> CREATE TABLE t1 AS
   ...>        SELECT 1 as a, 2 as b;
   sqlite> CREATE TABLE t2(c, d);
   sqlite> INSERT INTO t2 SELECT a, b FROM t1;
   sqlite> SELECT * FROM t2;
   1|2
   ```

2. Now, use the stacks of one dog to insert stacks of two dogs. It's possible to `INSERT INTO` a table rows selected from that same table. For example,

   ```
   sqlite> CREATE TABLE ints AS
      ...>   SELECT 1 AS n UNION
      ...>   SELECT 2      UNION
      ...>   SELECT 3;
   sqlite> INSERT INTO ints(n) SELECT n+3 FROM ints;
   sqlite> SELECT * FROM ints;
   1
   2
   3
   4
   5
   6
   ```

3. Repeat step 3 to create stacks of three dogs, then of four dogs.

Once you've built up to stacks of four dogs in your `stacks_helper` table, use it to fill in the `stacks` table!

Use Ok to test your code:

```
python3 ok -q stacks
```

In the solution, we follow the recommended procedure outlined in the problem above.

Here's some details to think about:

- Each iteration, we will generate the stack with n + 1 dogs, but we'll also regenerate all the previous stacks! For example, the stacks of size 1 are still around to generate the stacks of size 2. As such there are many duplicate rows of stack size 1, 2, and 3 in our `stack_helper`.
- This turns out not to be an issue: we got lucky since there weren't any stacks of size less than 4 that were tall enough. But even if there were, we could use `DISTINCT` to remove duplicate rows.
- Is there a way we could be more space efficient? Think about how we could generate new rows without keeping around *all* the previous ones.

Once we have everything in our `stack_helper` table, we just keep the rows from it that correspond to the tallest stacks. We also no longer need the `last_height` column.

Optional Scheme Questions

Q8: Dragon

Implement `dragon`, which draws a dragon curve (https://en.wikipedia.org/wiki/Dragon_curve). The strategy for how to draw a dragon curve is as follows. First create a list of instructions for how to draw the dragon curve. To do this, we start with the list `(f x)` and apply the following rewrite rules repeatedly

- `x -> (x r y f r)`
- `y -> (l f x l y)`

First implement `flatmap` function, which takes in a function and a list, and concatentates the result of mapping the function to every element of the list.

Then implement `expand`, which should implement the above rules in terms of `flatmap`

and then execute the interpreter on each argument by the following rules

- `x` or `y`: do nothing
- `f`: move forward by `dist`
- `l`: turn left 90 degrees
- `r`: turn right 90 degrees

We have given you a definition of `dragon` in terms of the `expand` and `interpret` functions. Complete these functions to see the dragon curve!

To learn how to control the turtle, please check out the scheme specification (/articles/scheme-builtins.html#turtle-graphics).

```
(define (flatmap f x)
  (define (h z x)
    (if (null? x)
        z
        (h (append z (f (car x))) (cdr x))))
  (h nil x))

(define (expand lst)
  (flatmap (lambda (x)
             (cond
               ((equal? x 'x) '(x r y f r))
               ((equal? x 'y) '(l f x l y))
               (else          (list x))))
           lst))

(define (interpret instr dist)
  (if (null? instr)
      nil
      (begin (define inst (car instr))
             (cond
               ((equal? 'f inst) (fd dist))
               ((equal? 'r inst) (rt 90))
               ((equal? 'l inst) (lt 90)))
             (interpret (cdr instr) dist))))

(define (apply-many n f x)
  (if (zero? n)
      x
      (apply-many (- n 1) f (f x))))

(define (dragon n d)
  (interpret (apply-many n expand '(f x)) d))
```

To test your `flatmap` and `expand` functions, run the following command.

Use Ok to test your code:

```
python3 ok -q dragon
```

To create a dragon curve or visually debug your code, run `(speed 0) (dragon 10 10)`. (The function `(speed 0)` makes the turtle move faster, if you don't do this it will take forever.)

Unfortunately, this will only run in the interpreter you launch with `python3 scheme`, so to test your code, run `python3 scheme -i lab11.scm` and then the command `(speed 0) (dragon 10 10)`.

> Hint: if you are getting a RecursionError, reimplement `flatmap` and `interpret` to be tail recursive.

CS 61A (/)

Weekly Schedule (/weekly.html)

Office Hours (/office-hours.html)

Staff (/staff.html)

Resources (/resources.html)

Studying Guide (/articles/studying.html)

Debugging Guide (/articles/debugging.html)

Composition Guide (/articles/composition.html)

Policies (/articles/about.html)

Assignments (/articles/about.html#assignments)

Exams (/articles/about.html#exams)

Grading (/articles/about.html#grading)