Instructions

> Please download homework materials `hw10.zip` from our QQ group if you don't have one.

In this homework, you are required to complete the problems described in section 2. The starter code for these problems is provided in `hw10.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 hw10.sql
   ```

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

   ```
   python sqlite_shell.py < hw10.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.

Creating Tables

You can create SQL tables either from scratch or from existing tables.

The following statement creates a table by specifying column names and values without referencing another table. Each `SELECT` clause specifies the values for one row, and `UNION` is used to join rows together. The `AS` clauses give a name to each column; it need not be repeated in subsequent rows after the first.

```
CREATE TABLE [table_name] AS
  SELECT [val1] AS [column1], [val2] AS [column2], ... UNION
  SELECT [val3]             , [val4]             , ... UNION
  SELECT [val5]             , [val6]             , ...;
```

Let's say we want to make the following table called `big_game` which records the scores for the Big Game each year. This table has three columns: `berkeley`, `stanford`, and `year`.

berkeley	stanford	year
30	7	2002
28	16	2003
17	38	2014

We could do so with the following `CREATE TABLE` statement:

```
CREATE TABLE big_game AS
  SELECT 30 AS berkeley, 7 AS stanford, 2002 AS year UNION
  SELECT 28,             16,            2003         UNION
  SELECT 17,             38,            2014;
```

Selecting From Tables

More commonly, we will create new tables by selecting specific columns that we want from existing tables by using a `SELECT` statement as follows:

```
SELECT [columns] FROM [tables] WHERE [condition] ORDER BY [columns] LIMIT [limit];
```

Let's break down this statement:

- `SELECT [columns]` tells SQL that we want to include the given columns in our output table; `[columns]` is a comma-separated list of column names, and `*` can be used to select all columns
- `FROM [table]` tells SQL that the columns we want to select are from the given table; see the *joins section* to see how to select from multiple tables
- `WHERE [condition]` filters the output table by only including rows whose values satisfy the given `[condition]`, a boolean expression
- `ORDER BY [columns]` orders the rows in the output table by the given comma-separated list of columns
- `LIMIT [limit]` limits the number of rows in the output table by the integer `[limit]`

Note: We capitalize SQL keywords purely because of style convention. It makes queries much easier to read, though they will still work if you don't capitalize keywords.

Here are some examples:

Select all the contents from the `big_game` table:

```
sqlite> SELECT * from big_game;
17|38|2014
28|16|2003
30|7|2002
```

Select all of Berkeley's scores from the `big_game` table, but only include scores from years past 2002:

```
sqlite> SELECT berkeley FROM big_game WHERE year > 2002;
28
17
```

Select the scores for both schools in years that Berkeley won:

```
sqlite> SELECT berkeley, stanford FROM big_game WHERE berkeley > stanford;
30|7
28|16
```

Select the years that Stanford scored more than 15 points:

```
sqlite> SELECT year FROM big_game WHERE stanford > 15;
2003
2014
```

SQL operators

Expressions in the `SELECT`, `WHERE`, and `ORDER BY` clauses can contain one or more of the following operators:

- comparison operators: `=`, `>`, `<`, `<=`, `>=`, `<>` or `!=` ("not equal")
- boolean operators: `AND`, `OR`
- arithmetic operators: `+`, `-`, `*`, `/`
- concatenation operator: `||`

Here are some examples:

Output the ratio of Berkeley's score to Stanford's score each year:

```
sqlite> SELECT berkeley * 1.0 / stanford FROM big_game;
0.447368421052632
1.75
4.28571428571429
```

Output the sum of scores in years where both teams scored over 10 points:

```
sqlite> SELECT berkeley + stanford FROM big_game WHERE berkeley > 10 AND stanford > 10;
55
44
```

Output a table with a single column and single row containing the value "hello world":

```
sqlite> SELECT "hello" || " " || "world";
hello world
```

Joins

To select data from multiple tables, we can use joins. There are many types of joins, but the only one we'll worry about is the inner join. To perform an inner join on two on more tables, simply list them all out in the `FROM` clause of a `SELECT` statement:

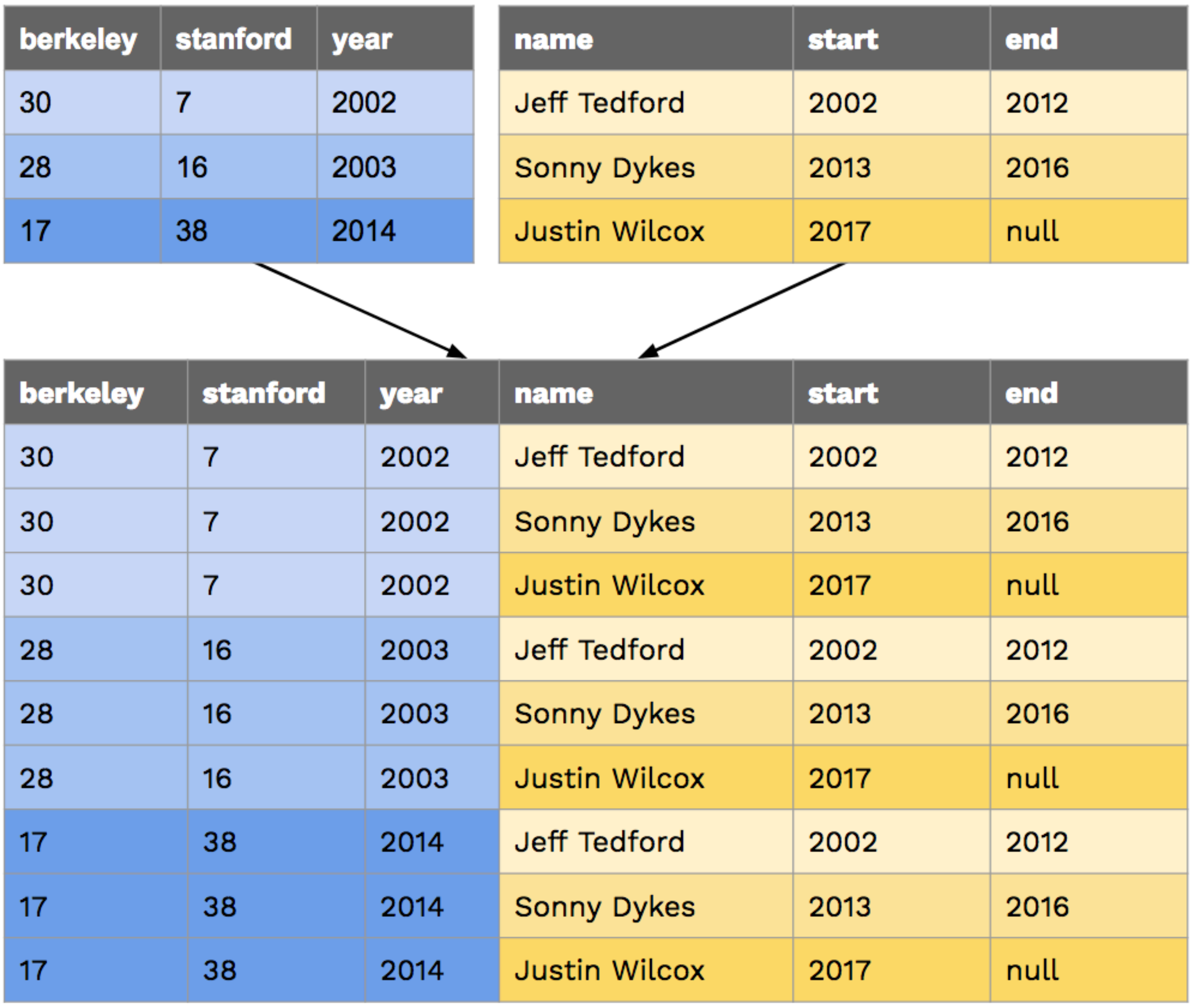
```
SELECT [columns] FROM [table1], [table2], ... WHERE [condition] ORDER BY [columns]
LIMIT [limit];
```

We can select from multiple different tables or from the same table multiple times.

Let's say we have the following table that contains the names of head football coaches at Cal since 2002:

```
CREATE TABLE coaches AS
  SELECT "Jeff Tedford" AS name, 2002 as start, 2012 as end UNION
  SELECT "Sonny Dykes"         , 2013         , 2016        UNION
  SELECT "Justin Wilcox"       , 2017         , null;
```

When we join two or more tables, the default output is a cartesian product (wiki or baidu). For example, if we joined `big_game` with `coaches`, we'd get the following:



berkeley	stanford	year
30	7	2002
28	16	2003
17	38	2014

name	start	end
Jeff Tedford	2002	2012
Sonny Dykes	2013	2016
Justin Wilcox	2017	null

berkeley	stanford	year	name	start	end
30	7	2002	Jeff Tedford	2002	2012
30	7	2002	Sonny Dykes	2013	2016
30	7	2002	Justin Wilcox	2017	null
28	16	2003	Jeff Tedford	2002	2012
28	16	2003	Sonny Dykes	2013	2016
28	16	2003	Justin Wilcox	2017	null
17	38	2014	Jeff Tedford	2002	2012
17	38	2014	Sonny Dykes	2013	2016
17	38	2014	Justin Wilcox	2017	null

If we want to match up each game with the coach that season, we'd have to compare columns from the two tables in the `WHERE` clause:

```
sqlite> SELECT * FROM big_game, coaches WHERE year >= start AND year <= end;
17|38|2014|Sonny Dykes|2013|2016
28|16|2003|Jeff Tedford|2002|2012
30|7|2002|Jeff Tedford|2002|2012
```

The following query outputs the coach and year for each Big Game win recorded in `big_game`:

```
sqlite> SELECT name, year FROM big_game, coaches
   ...>     WHERE berkeley > stanford AND year >= start AND year <= end;
Jeff Tedford|2003
Jeff Tedford|2002
```

In the queries above, none of the column names are ambiguous. For example, it is clear that the `name` column comes from the `coaches` table because there isn't a column in the `big_game` table with that name. However, if a column name exists in more than one of the tables being joined, or if we join a table with itself, we must disambiguate the column names using *aliases*.

For examples, let's find out what the score difference is for each team between a game in `big_game` and any previous games. Since each row in this table represents one game, in order to compare two games we must join `big_game` with itself:

```
sqlite> SELECT b.Berkeley - a.Berkeley, b.Stanford - a.Stanford, a.Year, b.Year
   ...>     FROM big_game AS a, big_game AS b WHERE a.Year < b.Year;
-11|22|2003|2014
-13|21|2002|2014
-2|9|2002|2003
```

In the query above, we give the alias `a` to the first `big_game` table and the alias `b` to the second `big_game` table. We can then reference columns from each table using dot notation with the aliases, e.g. `a.Berkeley`, `a.Stanford`, and `a.Year` to select from the first table.

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 hw10.sql
```

Or only starts with the original data in `hw10_data.sql`:

```
$ python sqlite_shell.py --init hw10_data.sql
```

Dog Data

In each question below, you will define a new table based on the following tables.

```
CREATE TABLE parents AS
  SELECT "abraham" AS parent, "barack" AS child UNION
  SELECT "abraham"          , "clinton"         UNION
  SELECT "delano"           , "herbert"         UNION
  SELECT "fillmore"         , "abraham"         UNION
  SELECT "fillmore"         , "delano"          UNION
  SELECT "fillmore"         , "grover"          UNION
  SELECT "eisenhower"       , "fillmore";

CREATE TABLE dogs AS
  SELECT "abraham" AS name, "long" AS fur, 26 AS height UNION
  SELECT "barack"         , "short"      , 52           UNION
  SELECT "clinton"        , "long"       , 47           UNION
  SELECT "delano"         , "long"       , 46           UNION
  SELECT "eisenhower"     , "short"      , 35           UNION
  SELECT "fillmore"       , "curly"      , 32           UNION
  SELECT "grover"         , "short"      , 28           UNION
  SELECT "herbert"        , "curly"      , 31;

CREATE TABLE sizes AS
  SELECT "toy" AS size, 24 AS min, 28 AS max UNION
  SELECT "mini"       , 28       , 35        UNION
  SELECT "medium"     , 35       , 45        UNION
  SELECT "standard"   , 45       , 60;
```

Your tables should still perform correctly even if the values in these tables change. For example, if you are asked to list all dogs with a name that starts with h, you should write:

```
SELECT name FROM dogs WHERE "h" <= name AND name < "i";
```

Instead of assuming that the `dogs` table has only the data above and writing

```
SELECT "herbert";
```

The former query would still be correct if the name `grover` were changed to `hoover` or a row was added with the name `harry`.

Problem 1: Size of Dogs (100 pts)

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`.

Create a `size_of_dogs` table with two columns, one for each dog's `name` and another for its `size`.

```sql
-- The size of each dog
CREATE TABLE size_of_dogs AS
  SELECT "REPLACE THIS LINE WITH YOUR SOLUTION";
```

The output should look like the following:

```
sqlite> select * from size_of_dogs;
abraham|toy
barack|standard
clinton|standard
delano|standard
eisenhower|mini
fillmore|mini
grover|toy
herbert|mini
```

Problem 2: By Parent Height (100 pts)

Create a table `by_parent_height` 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.

```
-- All dogs with parents ordered by decreasing height of their parent
CREATE TABLE by_parent_height AS
  SELECT "REPLACE THIS LINE WITH YOUR SOLUTION";
```

For example, `fillmore` has a parent (`eisenhower`) with height 35, and so should appear before `grover` who has a parent (`fillmore`) with height 32.

```
sqlite> select * from by_parent_height;
herbert
fillmore
abraham
delano
grover
barack
clinton
```

Hint: You might want to review the joins section.

Problem 3: Sentences (100 pts)

There are two pairs of siblings that have the same size. Create a table that contains a row with a string for each of these pairs. Each string should be a sentence describing the siblings by their size.

```
-- Sentences about siblings that are the same size
CREATE TABLE sentences AS
  SELECT "REPLACE THIS LINE WITH YOUR SOLUTION";
```

Each sibling pair should appear *only once* in the output *but in any order*, and siblings should be listed *in alphabetical order* (e.g. `"barack plus clinton..."` instead of `"clinton plus barack..."`), as follows:

```
sqlite> select * from sentences;
The two siblings, barack plus clinton have the same size: standard
The two siblings, abraham plus grover have the same size: toy
```

> *Hint1*: First, create a helper table containing each pair of siblings. This will make comparing the sizes of siblings when constructing the main table easier.
>
> *Hint2*: If you join a table with itself, use `AS` within the `FROM` clause to give each table an alias.
>
> *Hint3*: In order to concatenate two strings into one, use the `||` operator.
