

# Lab 13: SQL (Optional)

**lab13.zip (lab13.zip)**

*Due by 11:59pm on Monday, December 5.*

## Starter Files

Download lab13.zip (lab13.zip). Inside the archive, you will find starter files for the questions in this lab, along with a copy of the Ok (ok) autograder.

**This lab is optional. You do not need to submit it. All students will receive attendance credit.**

## Topics

Consult this section if you want 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

## SQL Basics

### 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 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 or 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
```

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 ([https://en.wikipedia.org/wiki/Cartesian\\_product](https://en.wikipedia.org/wiki/Cartesian_product)). For example, if we joined `big_game` with `coaches`, we'd get the following:

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

## 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 ([http://www.sqlite.org/lang\\_aggfunc.html](http://www.sqlite.org/lang_aggfunc.html)) such as `MAX(column)` combines the values in multiple rows.

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;
13
```

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
```

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
```

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).

## Troubleshooting

# 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 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 (<http://www.sqlite.org/download.html>).

**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
```

### Usage

## 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 or Git Bash with the following command:

```
python3 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: (Here we're using the `lab11.sql` file as an example.)

1. Runs your code and then exits SQLite immediately afterwards.

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

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



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

# Questions

## Q1: What Would SQL print?

Note: there is no submission for this question

First, load the tables into sqlite3.

```
$ python3 sqlite_shell.py --init lab13.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 `data.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 for all (
-----

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: Go Bears! (And Dogs?)

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...

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
blue|dog
blue|dog
blue|dog
blue|dog
blue|dog
blue|dog
blue|dog
```

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

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 blue_dog_songs;  
blue|dog|Glimpse of Us  
blue|dog|The Other Side of Paradise  
blue|dog|Leave the Door Open  
blue|dog|Bohemian Rhapsody  
blue|dog|Dancing Queen  
blue|dog|Palette  
blue|dog|Bohemian Rhapsody  
blue|dog|good 4 u  
blue|dog|Clair de Lune  
blue|dog|Dancing Queen
```

```
CREATE TABLE blue_dog_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!

Use Ok to test your code:

```
python3 ok -q blue_dog
```



### Q3: 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 "REPLACE THIS LINE WITH YOUR SOLUTION";
```

Use Ok to test your code:

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



## Q4: Matchmaker, Matchmaker

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. 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 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";
```

Use Ok to test your code:

```
python3 ok -q matchmaker
```



## Q5: Sevens

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.**

The first 10 lines of this table should look like this:

```
sqlite> SELECT * FROM sevens LIMIT 10;
7
I find this question condescending
7
seven
7
7
7
7
```

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

Use Ok to test your code:

```
python3 ok -q sevens
```



# Cyber Monday

## Q6: Price Check

After you are full from your Thanksgiving dinner, you realize that you still need to buy gifts for all your loved ones over the holidays. However, you also want to spend as little money as possible (you're not cheap, just looking for a great sale!).

Let's start off by surveying our options. Using the `products` table, write a query that creates a table `average_prices` that lists categories and the average price of items in the category (using MSRP ([https://en.wikipedia.org/wiki/List\\_price](https://en.wikipedia.org/wiki/List_price)) as the price).

You should get the following output:

```
sqlite> SELECT category, ROUND(average_price) FROM average_prices;
computer|109.0
games|350.0
phone|90.0
```

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

Use Ok to test your code:

```
python3 ok -q cyber-monday-part1
```



## Q7: The Price is Right

Now, you want to figure out which stores sell each item in `products` for the lowest price. Write a SQL query that uses the `inventory` table to create a table `lowest_prices` that lists items, the stores that sells that item for the lowest price, and the price that the store sells that item for.

You should expect the following output:

```
sqlite> SELECT * FROM lowest_prices;
Hallmart|GameStation|298.98
Targive|QBox|390.98
Targive|iBook|110.99
RestBuy|kBook|94.99
Hallmart|qPhone|85.99
Hallmart|rPhone|69.99
RestBuy|uPhone|89.99
RestBuy|wBook|114.29
```

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

Use Ok to test your code:

```
python3 ok -q cyber-monday-part2
```



## Q8: Bang for your Buck

You want to make a shopping list by choosing the item that is the best deal possible for every category. For example, for the "phone" category, the uPhone is the best deal because the MSRP price of a uPhone divided by its ratings yields the lowest cost. That means that uPhones cost the lowest money per rating point out of all of the phones. Note that the item with the lowest MSRP price may not necessarily be the best deal.

Write a query to create a table `shopping_list` that lists the items that you want to buy from each category.

After you've figured out which item you want to buy for each category, add another column that lists the store that sells that item for the lowest price.

You should expect the following output:

```
sqlite> SELECT * FROM shopping_list;
GameStation|Hallmart
uPhone|RestBuy
wBook|RestBuy
```

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

**Hint:** You should use the `lowest_prices` table you created in the previous question.

**Hint 2:** One approach to this problem is to create another table, then create `shopping_list` by selecting from that table.



Use Ok to test your code:

```
python3 ok -q cyber-monday-part3
```



## Q9: Driving the Cyber Highways

Using the Mb (megabits) column from the `stores` table, write a query to calculate the total amount of bandwidth needed to get everything in your shopping list.

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

**Hint:** You should use the `shopping_list` table you created in the previous question.

Use Ok to test your code:

```
python3 ok -q cyber-monday-part4
```



