01: Selecting Data

Why use SQL?

At it's heart, SQL is a relational database. Meaning that data is separated out into many tables, and a bunch of other really important things that we're totally not covering. However, using SQL still provides several advantages for working with data, even if you aren't going to go too deeply down the "relational database" road.

Using a SQL platform for your data management can provide benefits such as data protection and easier data querying over traditional spreadsheet applications. In this workshop, we're going to explore the basics of SQL to start getting comfortable with it, but also to help you better understand when you're in a situation that would benefit from the use of a SQL database.

What is SQLite?

There are many dialects of SQL, and SQLite is just one of them. It is free, light weight, and available all over the place. So a great place to get started! Don't worry, most of the skills you learn in SQLite will work with other forms of SQL.

What are queries?

Your data will live in tables within your SQL database. They're just sitting there, waiting for you to do stuff to them. You can edit the data depending on the platform you're using, but most of the time you'll be treating the data tables as a read-only data file.

This means that you'll be crafting queries that will be sent off to SQL, which will then process the query over our data, and return some results back to the system that we are using.

In the case of SQLite manager, to run queries:

1. Click on the "Execute SQL" tab.
2. Type some SQL into the "Enter SQL" text box
3. Click "Run SQL"
4. The bottom white space will now be populated with the results.

`SELECT` queries

The standard first query for any database is to do a select statement on everything.

```
SELECT * FROM pettigrew;
```

You should see all 601 rows of data in the results area.

This query is composed of several things:

- SQL keywords: SELECT, and FROM
- A wildcard (`*`) to represent "all columns"
- the table name: `pettigrew`
- Final puncutation (`;`) to indicate the query is done

Roughly speaking, this decomposes to "show me (columns) from (table name)".

Nearly all queries you'll run will work along these lines. We can add nuance to what we want to appear and where we want to get it, which is basically the bulk of the rest of this workshop.

Specify columns

`SELECT Date FROM pettigrew;` to see all contents of just the Date column.

Note: that my case needs to match for the table specific names of things, but doesn't matter for the SQL keywords.

```
SELECT BoxNumber, FolderNumber FROM pettigrew;
```

To select multiple columns, just place them all in separated by a comma.

Unique values

`DISTINCT` KEYWORD

An essential for data exploration, we often want to see the unique values for a single data column. We use the `DISTINCT` keyword to do this.

```
SELECT DISTINCT BoxNumber FROM pettigrew;
```

But this isn't really operating on single columns. In fact, it will give you all the distinct rows returned. Meaning that you can provide multiple columns and it will return the distinct pairs of data.

```
SELECT DISTINCT BoxNumber, FolderNumber FROM pettigrew;
```

This ends up returning 601 rows of data, meaning that these are naturally unique combinations.

Change it to `SELECT DISTINCT BoxNumber, Date FROM pettigrew;` and you get 558 rows, so you can tell that some folders share the same date pattern.

Filtering

`WHERE` keyword

Simple filtering is pretty decent to accomplish in normal spreadsheet programs, but complex filtering is much easier in SQL. We can specify logical conditions that the values of columns must satisf to be returned.

Say we want all the entries from box 1:

`SELECT * FROM pettigrew WHERE BoxNumber = 1;`.

The `WHERE` keyword comes in after your `FROM` section and includes a conditioanl check. Note that we use `=` to represent an equality statement.

You can also match strings here:

`SELECT * FROM pettigrew WHERE Date = 'n.d.';` will filter for every record with a date value of exactly `n.d.`.

Compound filtering with boolean keywords: `and` & `or`

You can combine multiple checks inside a single `WHERE` section.

```
SELECT * FROM pettigrew WHERE Date = 'n.d.' AND BoxNumber = 1;
```

```
SELECT * FROM pettigrew WHERE BoxNumber = 1 OR BoxNumber = 2;
```

Use `()` to group compound checks

Sometimes you need to have a bunch of filters in place and need to be explicit about the order of operations.

```
SELECT * FROM pettigrew WHERE (BoxNumber = 1 OR BoxNumber = 2) AND Date = 'n.d.';
```

Use the `IN` keyword to add multiple options

The previous example only had two possibilites for BoxNumber. We could use the `AND` keyword over and over, but we can simplify our query with the `IN ()` option.

```
SELECT * FROM pettigrew WHERE BoxNumber in (1, 2, 3, 4, 5) AND Date = 'n.d.';
```

This syntax makes it easy to add options in and out of the query.

Sorting

`ORDER BY` keyword

This keyword can be used to specify how to sort certain columns.

`SELECT * FROM pettigrew ORDER BY Date;` .

How numbers and letters are sorted as strings is the topic of another discussion. This query will sort the column Date in ascending (`asc`) order (by default), but you can specify `desc` to reverse it.

```
SELECT * FROM pettigrew ORDER BY Date DESC;
```

You can even sort by multiple columns:

```
SELECT * FROM pettigrew WHERE BoxNumber in(7, 8) ORDER BY Date, Contents;
```

But note that the order of sorting will be from left to right.

```
SELECT * FROM pettigrew WHERE BoxNumber in(7, 8) ORDER BY Contents, Date;
```

Challenge

What have we learned so far about our data from these queries?

Can we construct queries to confirm your observations?