WEEK-10 DAY-2
Data! Data! Data!

SQL Learning Objectives

SQL is the language of relational data. It is one of the core languages that most software developers know because of its prevalence in the industry due to the common use of RDBMSes. The objectives for your SQL learning journey cover:

- How to use the `SELECT ... FROM ...` statement to select data from a single table
- How to use the `WHERE` clause on `SELECT`, `UPDATE`, and `DELETE` statements to narrow the scope of the command
- How to use the `JOIN` keyword to join two (or more) tables together into a single virtual table
- How to use the `INSERT` statement to insert data into a table
- How to use an `UPDATE` statement to update data in a table
- How to use a `DELETE` statement to remove data from a table
- How to use a seed file to populate data in a database
- How to perform relational database design
- How to use transactions to group multiple SQL commands into one succeed or fail operation
- How to apply indexes to tables to improve performance
- Explain what and why someone would use `EXPLAIN`
- Demonstrate how to install and use the **node-postgres** library and its `Pool` class to query a PostgreSQL-managed database
- Explain how to write prepared statements with placeholders for parameters of the form "$1", "$2", and so on

Retrieving Rows From A Table Using SELECT

In the first reading, we covered SQL and PostgreSQL and how to set up PostgreSQL. In this reading, we're going to learn how to write a simple SQL query using SELECT.

What is a query?

SQL stands for *Structured Query Language*, and whenever we write SQL we're usually querying a database. A query is simply a question we're asking a database, and we're aiming to get a response back. The response comes back to us as a list of table rows.

Example table

Let's say we had the following database table called `puppies`. We'll use this table to make our queries:

puppies table

name	age_yrs	breed	weight_lbs	microchipped
Cooper	1	Miniature Schnauzer	18	yes
Indie	0.5	Yorkshire Terrier	13	yes
Kota	0.7	Australian Shepherd	26	no
Zoe	0.8	Korean Jindo	32	yes
Charley	1.5	Basset Hound	25	no
Ladybird	0.6	Labradoodle	20	yes
Callie	0.9	Corgi	16	no
Jaxson	0.4	Beagle	19	yes
Leinni	1	Miniature Schnauzer	25	yes
Max	1.6	German Shepherd	65	no

Using psql in the terminal

As we covered in the first reading, psql allows us to access the PostgreSQL server and make queries via the terminal. Open up the terminal on your machine, and connect to the PostgreSQL server by using the following psql command:

```
psql -U postgres
```

The above command lets you access the PostgreSQL server as the user 'postgres' (`-U` stands for user). After you enter this command, you'll be prompted for the password that you set for the 'postgres' user during installation. Type it in, and hit Enter. Once you've successfully logged in, you should see the following in the terminal:

```
Password for user postgres:
psql (11.5, server 11.6)
Type "help" for help.

postgres=#
```

You can exit psql at anytime with the command `\q`, and you can log back in with `psql -U postgres`. (*See this Postgres Cheatsheet for a list of more PSQL commands.*)

We'll use the following PostgreSQL to create the `puppies` table above. After you've logged into the psql server, type the following code and hit Enter.

puppies.sql

```
create table puppies (
  name VARCHAR(100),
  age_yrs NUMERIC(2,1),
  breed VARCHAR(100),
  weight_lbs INT,
  microchipped BOOLEAN
);

insert into puppies
values
('Cooper', 1, 'Miniature Schnauzer', 18, 'yes');

insert into puppies
values
('Indie', 0.5, 'Yorkshire Terrier', 13, 'yes'),
('Kota', 0.7, 'Australian Shepherd', 26, 'no'),
('Zoe', 0.8, 'Korean Jindo', 32, 'yes'),
('Charley', 1.5, 'Basset Hound', 25, 'no'),
('Ladybird', 0.6, 'Labradoodle', 20, 'yes'),
('Callie', 0.9, 'Corgi', 16, 'no'),
('Jaxson', 0.4, 'Beagle', 19, 'yes'),
('Leinni', 1, 'Miniature Schnauzer', 25, 'yes' ),
('Max', 1.6, 'German Shepherd', 65, 'no');
```

In the above SQL, we created a new table called `puppies`, and we gave it the following columns: `name`, `age_yrs`, `breed`, `weight_lbs`, and `microchipped`. We filled the table with ten rows containing data for each puppy, by using `insert into puppies values ()`.

We used the following PostgreSQL data types: `VARCHAR`, `NUMERIC`, `INT`, and `BOOLEAN`.

- `VARCHAR(n)` is a variable-length character string that lets you store up to *n* characters. Here we've set the character limit to 100 for the `name` and `breed` columns.
- `NUMERIC(p,s)` is a floating-point number with *p* digits and *s* number of places after the decimal point. Here we've set the values for the `age_yrs` column to up to two digits before the decimal and one place after the decimal.
- `INT` is a 4-byte integer, which we've set on the `weight_lbs` column.
- `BOOLEAN` is, of course, a Boolean value. We've set the `microchipped` column to accept Boolean values. SQL accepts the standard Boolean values `true`, `false`, or `null`. However, you'll note that we've used `yes` and `no` in our `microchipped` column because PostgreSQL Booleans can be any of the following values:

TRUE	FALSE
true	false
't'	'f'
'true'	'false'
'yes'	'no'
'y'	'n'
'1'	'0'

Simple SELECT Query

We can write a simple SELECT query to get results back from the table above. The syntax for the SELECT query is `SELECT [columns] FROM [table]`.

SELECT all rows

Using `SELECT *` is a quick way to get back all the rows in a given table. It is discouraged in queries that you write for your applications. Use it only when playing around with data, not for production code.

```
SELECT *
FROM   puppies;
```

Type the query above into your psql terminal, and make sure to add a semicolon at the end, which terminates the statement. `SELECT` and `FROM` should be capitalized. The above query should give us back the entire `puppies` table:

name	age_yrs	breed	weight_lbs	microchipped
Cooper	1	Miniature Schnauzer	18	yes
Indie	0.5	Yorkshire Terrier	13	yes
Kota	0.7	Australian Shepherd	26	no
Zoe	0.8	Korean Jindo	32	yes
Charley	1.5	Basset Hound	25	no

name	age_yrs	breed	weight_lbs	microchipped
Ladybird	0.6	Labradoodle	20	yes
Callie	0.9	Corgi	16	no
Jaxson	0.4	Beagle	19	yes
Leinni	1	Miniature Schnauzer	25	yes
Max	1.6	German Shepherd	65	no

SELECT by column

We can see all the rows in a given column by using `SELECT [column name]`.

```
SELECT name
FROM   puppies;
```

Type the query above into your psql terminal, and make sure to add a semicolon at the end, which terminates the statement. `SELECT` and `FROM` should be capitalized. The above query should give us back the following:

name
Cooper
Indie
Kota
Zoe
Charley
Ladybird
Callie
Jaxson
Leinni
Max

SELECT multiple columns

To see multiple columns, we can concatenate the column names by using commas between column names.

```
SELECT name
     , age_yrs
     , weight_lbs
FROM   puppies;
```

Type the query above into your psql terminal, and make sure to add a semicolon at the end, which terminates the statement. `SELECT` and `FROM` should be capitalized. The above query should give us back the following:

name	age_yrs	weight_lbs
Cooper	1	18
Indie	0.5	13
Kota	0.7	26
Zoe	0.8	32
Charley	1.5	25
Ladybird	0.6	20
Callie	0.9	16
Jaxson	0.4	19
Leinni	1	25
Max	1.6	65

Formatting SELECT statements

This is another of those hot-button topics with software developers. Some people like to put all the stuff on one line for each SQL keyword.

```
SELECT name, age_yrs, weight_lbs
FROM   puppies;
```

That works for short lists. But some tables have hundreds of columns. That gets long.

Some developers like what you saw earlier, the "each column name on its own line with the comma at the front".

```
SELECT name
     , age_yrs
     , weight_lbs
FROM   puppies;
```

They like this because if they need to comment out a column name, they can just put a couple of dashes at the beginning of the line.

```
SELECT name
--     , age_yrs
     , weight_lbs
FROM   puppies;
```

Some developers just do a word wrap when lines get too long.

All of these are fine. Just stay consistent within a project how you do them.

What we learned:

- What a query is
- How to connect to the PostgreSQL server with psql
- How to construct an example SQL table
- PostgreSQL data types
- How to write a simple SELECT query
- How to SELECT all rows, rows by column, and rows by multiple columns

---

Selecting Table Rows Using WHERE And Common Operators

In the last reading, we learned how to create a simple SQL query using SELECT. In this reading, we'll be adding a WHERE clause to our SELECT statement to further filter a database table and get specific rows back.

Using SELECT and WHERE

Previously, we covered how to use SELECT queries to fetch all of a table's rows or specified table rows by column(s). We can filter information returned by our query by using a WHERE clause in our SELECT statement.

Let's look at some examples of adding a WHERE clause using our `puppies` table from before:

name	age_yrs	breed	weight_lbs	microchipped
Cooper	1	Miniature Schnauzer	18	yes
Indie	0.5	Yorkshire Terrier	13	yes
Kota	0.7	Australian Shepherd	26	no
Zoe	0.8	Korean Jindo	32	yes
Charley	1.5	Basset Hound	25	no
Ladybird	0.6	Labradoodle	20	yes
Callie	0.9	Corgi	16	no
Jaxson	0.4	Beagle	19	yes
Leinni	1	Miniature Schnauzer	25	yes
Max	1.6	German Shepherd	65	no

WHERE clause for a single value

The simplest WHERE clause finds a row by a single column value. See the example below, which finds the rows where the breed equals 'Corgi':

```
SELECT name, breed FROM puppies
  WHERE breed = 'Corgi';
```

`SELECT`, `FROM`, and `WHERE` are capitalized. Notice that the string must use single quotation marks. *Note: PostgreSQL converts all names of tables, columns, functions, etc. to lowercase unless they're double quoted.* For example: `create table Foo()` will create a table called `foo`, and `create table "Bar"()` will create a table called `Bar`. If you use double quotation marks in

the query above, you'll get an error that says `column "Corgi" does not exist` because it thinks you're searching for the capitalized column name `Corgi`.

Use the command `psql -U postgres` in the terminal, and type in your 'postgres' user password to connect to the PostgreSQL server. You should have a puppies table in your postgres database from the last reading. Once you're in psql, enter the query above into the terminal and press Enter. You should get back one result for Callie the Corgi.

```
  name  | breed
--------+-------
 Callie | Corgi
```

WHERE clause for a list of values

We can also add a WHERE clause to check for a list of values. The syntax is `WHERE [column] IN ('value1', 'value2', 'value3')`. Let's say we wanted to find the name and breed of the puppies who are Corgis, Beagles, or Yorkshire Terriers. We could do so with the query below:

```
SELECT name, breed FROM puppies
  WHERE breed IN ('Corgi', 'Beagle', 'Yorkshire Terrier');
```

Entering this query into psql should yield the following results:

```
  name  |       breed
--------+-------------------
 Indie  | Yorkshire Terrier
 Callie | Corgi
 Jaxson | Beagle
```

WHERE clause for a range of values

In addition to checking for string values, we can use the WHERE clause to check for a range of numeric/integer values. This time, let's find the name, breed, and age of the puppies who are between 0 to 6 months old.

```
SELECT name, breed, age_yrs FROM puppies
  WHERE age_yrs BETWEEN 0 AND 0.6;
```

Entering this query into psql should yield the following results:

```
   name   |       breed       | age_yrs
----------+-------------------+---------
 Indie    | Yorkshire Terrier |     0.5
 Ladybird | Labradoodle       |     0.6
 Jaxson   | Beagle            |     0.4
```

ORDER BY

Getting the values back from a database in any order it wants to give them to you is ludicrous. Instead, you will often want to specify the order in which you get them back. Say you wanted them in alphabetical order by their name. Then, you would write

```
SELECT name, breed
FROM puppies
ORDER BY name;
```

Say you wanted that returned from oldest dog to youngest dog. You would write

```
SELECT name, breed
FROM puppies
ORDER BY age_yrs DESC;
```

where `DESC` means in descending order. Note that the column that you order on does not have to appear in the column list of the `SELECT` statement.

LIMIT and OFFSET

Say your query would return one million rows because you've cataloged every puppy in the world. That would be a lot for any application to handle. Instead, you may want to limit the number of rows returned. You can do that with the `LIMIT` keyword.

```
SELECT name, breed
FROM puppies
ORDER BY age_yrs
LIMIT 100;
```

That would return the name and breed of the 100 youngest puppies. (Why?) That is, of the million rows that the statement would find, it *limits* the number to

only 100.

Let's say you want to see the *next* 100 puppies after the first hundred. You can do that with the `OFFSET` keyword which comes after the `LIMIT` clause.

```
SELECT name, breed
FROM puppies
ORDER BY age_yrs
LIMIT 100 OFFSET 100;
```

That will return only rows 101 - 200 of the result set. It *limits* the total number of records to return to 100. Then, it starts at the 100th row and counts 100 records. Those are the records returned.

SQL operators

A SQL operator is a word or character that is used inside a WHERE clause to perform comparisons or arithmetic operations. In the three examples above, we used SQL operators inside of WHERE clauses to filter table rows -- `=`, `IN`, `BETWEEN`, and `AND`.

The following is a listing of SQL operators. We can combine any of these operators in our query or use a single operator by itself.

Logical operators

Operator	Description
ALL	TRUE if all of the subquery values meet the condition.
AND	TRUE if all the conditions separated by AND are TRUE.
ANY	TRUE if any of the subquery values meet the condition.
BETWEEN	TRUE if the operand is within the range of comparisons.
EXISTS	TRUE if the subquery returns one or more records.
IN	TRUE if the operand is equal to one of a list of expressions.
LIKE	TRUE if the operand matches a pattern (accepts "wildcards").
NOT	Displays a record if the condition(s) is NOT TRUE.
OR	TRUE if any of the conditions separated by OR is TRUE.
SOME	TRUE if any of the subquery values meet the condition.

Here is another example query with a WHERE clause using several logical operators: `NOT`, `IN`, `AND`, and `LIKE`.

```
SELECT name, breed FROM puppies
  WHERE breed NOT IN ('Miniature Schnauzer', 'Basset Hound', 'Labradoodle')
    AND breed NOT LIKE '%Shepherd';
```

Note: Pay attention to that `LIKE` operator. You will use it more than you want to. The wildcard it uses is the percent sign. Here's a table to help you understand.

`LIKE`	Matches "dog"	Matches "hotdog"	Matches "dog-tired"	Matches "ordogordo"
`'dog'`	yes	no	no	no
`'%dog'`	yes	yes	no	no
`'dog%'`	yes	no	yes	no
`'%dog%'`	yes	yes	yes	yes

Entering this query into psql should yield the following results:

```
  name  |       breed
--------+-------------------
 Indie  | Yorkshire Terrier
 Zoe    | Korean Jindo
 Callie | Corgi
 Jaxson | Beagle
```

With the query above, we filtered out six puppies: two Miniature Schnauzers, one Basset Hound, one Labradoodle, and two Shepherds. We started with ten puppies in the table, so we're left with four table rows. There are two puppies who are Shepherd breeds: an Australian Shepherd and a German Shepherd. We used the `LIKE` operator to filter these. In `'%Shepherd`, the `%` matches any substring value before the substring 'Shepherd'.

Arithmetic operators

Operator	Meaning	Syntax
+	Addition	a + b
-	Subtraction	a - b
*	Multiplication	a * b
/	Division	a / b
%	Modulus (returns remainder)	a % b

Here is an example query with a WHERE clause using the multiplication operator to find puppies that are 6 months old:

```
SELECT name, breed, age_yrs FROM puppies
  WHERE age_yrs * 10 = 6;
```

Entering the above query into psql will yield one result:

```
   name   |    breed    | age_yrs
----------+-------------+---------
 Ladybird | Labradoodle |     0.6
```

Comparison operators

Operator	Meaning	Syntax
=	Equals	a = b
!=	Not equal to	a != b
<>	Not equal to	a <> b
Greater than		a > b
<	Less than	a < b
>=	Greater than or equal to	a >= b
<=	Less than or equal to	a <= b
!<	Not less than	a !< b
!>	Not greater than	a !> b

Here is an example query with a WHERE clause using the `>` comparison operator:

```
SELECT name, breed, weight_lbs FROM puppies
  WHERE weight_lbs > 50;
```

Entering the above query into psql will yield one result:

```
 name |      breed      | weight_lbs
------+-----------------+------------
 Max  | German Shepherd |         65
```

What we learned:

- How to use a WHERE clause in a SELECT query
- How to construct WHERE clauses matching a single value, a list of values, and a range of values
- What SQL operators are
- How to use logical operators, arithmetic operators, and comparison operators

---

Inserting Data Into A Table

If you have data, but it's not in tables, does the data even exist? Not to an app! We often need to create relational databases on the back end of the Web apps we're building so that we can ultimately display this data on the front end of our application. All relational database data is stored in tables, so it's important to learn how to create tables and successfully query them.

Of the four data manipulation statements, `INSERT` is the easiest.

Create a new database named "folks". Now, create a new table named "friends" with the following column specifications.

Name	Data type	Constraints
id	`SERIAL`	`PRIMARY KEY`
first_name	`VARCHAR(255)`	`NOT NULL`
last_name	`VARCHAR(255)`	`NOT NULL`

Now that we have a new table, we need to add table rows with some data. We can insert a new table row using the following syntax:

```
INSERT INTO table_name
VALUES
  (column1_value, colum2_value, column3_value);
```

Let's fill out our "friends" table with information about five friends. In psql, enter the following to add new table rows. *Note the use of single quotation marks for string values. Also note that, since we used the* `SERIAL` *pseudo-type to auto-increment the ID values, we can simply write* `DEFAULT` *for the ID values when inserting new table rows.*

```
INSERT INTO friends (id, first_name, last_name)
VALUES
  (DEFAULT, 'Amy', 'Pond');
```

You can also completely omit the `DEFAULT` keyword if you specify the names of the columns that you want to insert into.

You can also use the "multiple values" insert. This prevents you from having to write `INSERT` with every statement. Even better, if one fails, they all fail. That can help protect your data integrity.

```
INSERT INTO friends (first_name, last_name)
VALUES
('Rose', 'Tyler'),
('Martha', 'Jones'),
('Donna', 'Noble'),
('River', 'Song');
```

Use `SELECT * FROM friends;` to verify that there are rows in the "friends" table:

```
appacademy=# SELECT * FROM friends;
 id | first_name | last_name
----+------------+-----------
  1 | Amy        | Pond
  2 | Rose       | Tyler
  3 | Martha     | Jones
  4 | Donna      | Noble
  5 | River      | Song
```

Now let's try to insert a new row using the ID of 5:

```
INSERT INTO friends (id, first_name, last_name)
VALUES (5, 'Jenny', 'Who');
```

Because ID is a primary key and that ID is already taken, we should get a message in psql that it already exists:

```
appacademy=# insert into friends values (5, 'Jenny', 'Who');
ERROR:  duplicate key value violates unique constraint "friends_pkey"
DETAIL:  Key (id)=(5) already exists.
```

What we learned:

- How to connect to an existing PostgreSQL database
- How to create a new PostgreSQL database
- How to create a new database table
- Accepted PostgreSQL data types
- How to add new rows to a database table
- What a primary key is/does

Foreign Keys And The JOIN Operation

In relational databases, *relationships* are key. We can create relationships, or *associations*, among tables so that they can access and share data. In a SQL database, we create table associations through *foreign keys* and *primary keys*.

You've learned about primary and foreign keys. Now, it's time to put them to use.

Setting up the database

Create a new database called "learn_joins". Connect to that database. Run the following SQL statements to create tables and the data in them.

```
CREATE TABLE breeds (
  id SERIAL,
  name VARCHAR(50) NOT NULL,
  PRIMARY KEY (id)
);
```

```
INSERT INTO breeds (name)
VALUES
('Australian Shepherd'),
('Basset Hound'),
('Beagle'),
('Corgi'),
('German Shepherd'),
('Korean Jindo'),
('Labradoodle'),
('Miniature Schnauzer'),
('Yorkshire Terrier');
```

```
CREATE TABLE puppies (
  id SERIAL,
  name VARCHAR(50) NOT NULL,
  age_yrs NUMERIC(3,1) NOT NULL,
  breed_id INTEGER NOT NULL,
  weight_lbs INTEGER NOT NULL,
  microchipped BOOLEAN NOT NULL DEFAULT FALSE,
  PRIMARY KEY(id),
  FOREIGN KEY (breed_id) REFERENCES breeds(id)
);
```

```
INSERT INTO puppies (name, age_yrs, breed_id, weight_lbs, microchipped)
VALUES
('Cooper', 1, 8, 18, true),
('Indie', 0.5, 9, 13, true),
('Kota', 0.7, 1, 26, false),
('Zoe', 0.8, 6, 32, true),
('Charley', 1.5, 2, 25, false),
('Ladybird', 0.6, 7, 20, true),
('Callie', 0.9, 4, 16, false),
('Jaxson', 0.4, 3, 19, true),
('Leinni', 1, 8, 25, true),
('Max', 1.6, 5, 65, false);
```

Using JOIN to retrieve rows from multiple tables

Now that we've set up an association between the "puppies" table and the "friends" table, we can access data from both tables. We can do so by using a JOIN operation in our SELECT query. Type the following into psql:

```
SELECT * FROM puppies
INNER JOIN breeds ON (puppies.breed_id = breeds.id);
```

The `JOIN` operation above is joining the "puppies" table with the "breeds" table together into a single table using the foreign key/primary key shared between the two tables: `breed_id`.

You should get all rows back containing all information for the puppies and breeds with matching `breed_id` values:

```
postgres=# SELECT * FROM puppies
postgres-# INNER JOIN breeds ON (puppies.breed_id = breeds.id);
 id |   name   | age_yrs | breed_id | weight_lbs | microchipped | id |        name
----+----------+---------+----------+------------+--------------+----+--------------------
  1 | Cooper   |     1.0 |        8 |         18 | t            |  8 | Miniature Schnauzer
  2 | Indie    |     0.5 |        9 |         13 | t            |  9 | Yorkshire Terrier
  3 | Kota     |     0.7 |        1 |         26 | f            |  1 | Australian Shepherd
  4 | Zoe      |     0.8 |        6 |         32 | t            |  6 | Korean Jindo
  5 | Charley  |     1.5 |        2 |         25 | f            |  2 | Basset Hound
  6 | Ladybird |     0.6 |        7 |         20 | t            |  7 | Labradoodle
  7 | Callie   |     0.9 |        4 |         16 | f            |  4 | Corgi
  8 | Jaxson   |     0.4 |        3 |         19 | t            |  3 | Beagle
  9 | Leinni   |     1.0 |        8 |         25 | t            |  8 | Miniature Schnauzer
 10 | Max      |     1.6 |        5 |         65 | f            |  5 | German Shepherd
(10 rows)
```

We could make our query more specific by selecting specific columns, adding a `WHERE` clause, or doing any number of operations that we could do in a normal `SELECT` query. Aside from an `INNER JOIN`, we could also do different types of `JOIN` operations. (Refer to this overview on PostgreSQL JOINS for more information.)

What we learned:

- What a foreign key is/does
- How to create a foreign key
- How to alter/update an existing table
- How to use the JOIN operation to get rows from two tables

Helpful links:

- PostgreSQL Docs: Constraints
- PostgreSQL Docs: Data Types > Numeric Types
- PostgreSQL Docs: Joins Between Tables
- PostgreSQL Tutorial: PostgreSQL Joins

Writing And Running A Seed File In PSQL

After a database is created, we need to populate it, or *seed* it, with data. Until now, we've used the command-line psql interface to create tables and insert rows into those tables. While that's fine for small datasets, it would be cumbersome to add a large dataset using the command line.

In this reading, we'll learn how to create and run a seed file, which makes the process of populating a database with test data much easier.

Creating a seed file

You can create a seed file by opening up VSCode or any text editor and saving a file with the `.sql` extension.

Let's create a seed file called `seed-data.sql` that's going to create a new table called `pies` and insert 50 pie rows into the table. Use the code below to create the seed file, and make sure to save your seed file on your machine.

seed-data.sql

```
CREATE TABLE pies (
  flavor VARCHAR(255) PRIMARY KEY,
  price FLOAT
);

INSERT INTO pies VALUES('Apple', 19.95);
INSERT INTO pies VALUES('Caramel Apple Crumble', 20.53);
INSERT INTO pies VALUES('Blueberry', 19.31);
INSERT INTO pies VALUES('Blackberry', 22.86);
INSERT INTO pies VALUES('Cherry', 22.32);
INSERT INTO pies VALUES('Peach', 20.45);
INSERT INTO pies VALUES('Raspberry', 20.99);
INSERT INTO pies VALUES('Mixed Berry', 21.45);
INSERT INTO pies VALUES('Strawberry Rhubarb', 24.81);
INSERT INTO pies VALUES('Banana Cream', 18.66);
INSERT INTO pies VALUES('Boston Toffee', 25.00);
INSERT INTO pies VALUES('Banana Nutella', 22.12);
INSERT INTO pies VALUES('Bananas Foster', 24.99);
INSERT INTO pies VALUES('Boston Cream', 23.75);
INSERT INTO pies VALUES('Cookies and Cream', 18.27);
INSERT INTO pies VALUES('Coconut Cream', 22.89);
INSERT INTO pies VALUES('Chess', 25.00);
INSERT INTO pies VALUES('Lemon Chess', 25.00);
INSERT INTO pies VALUES('Key Lime', 19.34);
INSERT INTO pies VALUES('Lemon Meringue', 19.58);
INSERT INTO pies VALUES('Guava', 18.92);
INSERT INTO pies VALUES('Mango', 19.55);
INSERT INTO pies VALUES('Plum', 20.21);
INSERT INTO pies VALUES('Apricot', 20.55);
INSERT INTO pies VALUES('Gooseberry', 23.54);
INSERT INTO pies VALUES('Lingonberry', 24.35);
INSERT INTO pies VALUES('Pear Vanilla Butterscotch', 25.13);
INSERT INTO pies VALUES('Baked Alaska', 25.71);
INSERT INTO pies VALUES('Chocolate', 19.00);
INSERT INTO pies VALUES('Chocolate Mousse', 21.46);
INSERT INTO pies VALUES('Mexican Chocolate', 28.33);
INSERT INTO pies VALUES('Chocolate Caramel', 26.67);
INSERT INTO pies VALUES('Chocolate Chip Cookie Dough', 18.65);
INSERT INTO pies VALUES('Pecan', 26.33);
INSERT INTO pies VALUES('Bourbon Caramel Pecan', 27.88);
INSERT INTO pies VALUES('Chocolate Pecan', 27.63);
INSERT INTO pies VALUES('Pumpkin', 20.91);
INSERT INTO pies VALUES('Sweet Potato', 20.75);
INSERT INTO pies VALUES('Purple Sweet Potato', 26.34);
INSERT INTO pies VALUES('Cheesecake', 28.99);
INSERT INTO pies VALUES('Butterscotch Praline', 29.78);
INSERT INTO pies VALUES('Salted Caramel', 30.32);
INSERT INTO pies VALUES('Salted Honey', 59.00);
INSERT INTO pies VALUES('Sugar Cream', 33.89);
INSERT INTO pies VALUES('Mississippi Mud', 28.67);
INSERT INTO pies VALUES('Turtle Fudge', 30.58);
INSERT INTO pies VALUES('Fruit Loops', 20.45);
INSERT INTO pies VALUES('Black Forest', 34.99);
INSERT INTO pies VALUES('Maple Cream', 35.88);
INSERT INTO pies VALUES('Smores', 26.43);
INSERT INTO pies VALUES('Milk Bar', 46.00);

SELECT * FROM pies;
```

Populating a database via < (“left caret”)

Now that you have a seed file, you can insert it into a database with a simple command.

Create a database named "bakery".

The syntax is `psql -d [database] < [path_to_file/file.sql]`. The left caret (`<`) takes the standard input from the file on the right (your seed file) and inputs it into the program on the left (`psql`).

Open up your terminal, and enter the following command. Make sure to replace `path_to_my_file` with the actual file path.

```
psql -d bakery < path_to_my_file/seed-data.sql
```

In the terminal, you should see a bunch of `INSERT` statements and the entire “pies” table printed out (from the `SELECT *` query in the seed file).

You can log into psql and use `\dt` to verify that your new table has been added to the database:

```
postgres=# \dt
List of relations
 Schema |     Name     | Type  |  Owner
 public | breeds       | table | appacademy
 public | pies         | table | appacademy
 public | puppies      | table | appacademy
```

Populating the database via | (“pipe”)

You could also use the “pipe” (`|`) to populate the database with your seed file.

The syntax is `cat [path_to_file/file.sql] | psql -d [database]`. 'cat' is a standard Unix utility that reads files sequentially, writing them to standard output. The "pipe" (`|`) takes the standard output of the command on the left and pipes it as standard input to the command on the right.

Try out this method in your terminal. If you have an existing "pies" table, you'll need to drop this table before you can add it again:

```
DROP TABLE pies;
```

Then, enter the following. Make sure to replace `path_to_my_file` with the actual file path.

```
cat path_to_my_file/seed-data.sql | psql -d postgres
```

Again, you should see a bunch of `INSERT` statements and the entire "pies" table printed out (from the `SELECT *` query in the seed file).

You can log into psql and use `\dt` to verify that your new table has been added to the database:

```
postgres=# \dt
List of relations
 Schema |     Name     | Type  |  Owner
 public | friends      | table | postgres
 public | pies         | table | postgres
 public | puppies      | table | postgres
```

What we learned:

- What a seed file is
- How to create a seed file
- How to populate a database with a seed file using <
- How to populate a database with a seed file using |

Create And Seed A Database Project

In our SQL readings, we installed PostgreSQL, and we learned how to create a new PostgreSQL database and how to create and run a seed file in our database.

In this project, you'll practice seeding your database with a seed file that's full of, well, *seeds*! Hopefully when you're done, you'll have *grown* your SQL knowledge.

Project overview

Practice creating a new database and piping a seed file into your database.

- In Phase 1, you'll create a new database.
- In Phase 2, you'll create a new seed data file.
- In Phase 3, you'll pipe your seed data in your database via `psql` on the command line.

Phase 1: Create a new database

First, log into `psql` on the command line as your user.

Second, create a new database in PostgreSQL called `farm` by using the following syntax:

```
CREATE DATABASE [databasename];
```

Note: You can check to make sure you've created a new database by viewing the list of databases with `\l`.

Phase 2: Create a seeds seed file

Create a seed file full of seeds! Set up a SQL file with seed data that will produce two tables: an "edible_seeds" with 40 rows and a "flower_seeds" table with 20 rows.

Edible Seeds

id	name	type	price_per_pound	in_stock
1	Chia	Pseudocereal	6.95	yes

id	name	type	price_per_pound	in_stock
2	Flax	Pseudocereal	6.90	yes
3	Amaranth	Pseudocereal	14.99	yes
4	Quinoa	Pseudocereal	12.49	no
5	Sesame	Pseudocereal	13.49	yes
6	Hemp	Other	10.99	yes
7	Chickpea	Legume	7.99	yes
8	Pea	Legume	7.50	no
9	Soybean	Legume	12.99	yes
10	Acorn	Nut	11.99	yes
11	Almond	Nut	13.99	yes
12	Beech	Nut	10.94	yes
13	Chestnut	Nut	13.99	yes
14	Water Chestnut	Nut	9.99	no
15	Macadamia	Nut	25.00	yes
16	Pistachio	Nut	20.00	yes
17	Pine nuts	Nut-like gymnosperm	23.00	yes
18	Pecan	Nut	6.99	yes
19	Juniper berries	Nut-like gymnosperm	11.99	yes
20	Cashew	Nut	23.61	yes
21	Hazelnut	Nut	25.49	yes
22	Sunflower seed	Other	9.99	yes
23	Poppy seed	Other	12.99	yes
24	Barley	Cereal	9.99	yes
25	Maize	Cereal	6.98	yes
26	Oats	Cereal	9.99	yes
27	Rice	Cereal	7.90	yes
28	Rye	Cereal	9.87	yes
29	Spelt	Cereal	7.50	yes
30	Wheat berries	Cereal	2.50	no
31	Buckwheat	Pseudocereal	5.60	yes
32	Jackfruit	Other	15.00	yes
33	Durian	Other	8.26	no
34	Lotus seed	Other	12.99	yes
35	Ginko	Nut-like gymnosperm	12.80	yes
36	Peanut	Legume	5.99	yes
37	Pumpkin seed	Other	5.99	yes
38	Watermelon seed	Other	6.99	yes
39	Pomegranate seed	Other	27.63	yes
40	Cacao bean	Other	12.99	yes

Use the following data types for your "edible_seeds" columns:

- id - SERIAL
- name - VARCHAR that accepts 255 characters
- type - VARCHAR that accepts 255 characters
- price_per_pound - FLOAT or REAL
- in_stock - BOOLEAN

Flower Seeds

id	name	main_color	seeds_per_packet	price_per_packet	in_stock
1	Begonia Fiona Red	Red	25	4.95	yes
2	Moonflower Seeds	White	25	2.95	yes

id	name	main_color	seeds_per_packet	price_per_packet	in_stock
3	Easy Wave F1 Lavender Sky Blue Petunia Seeds	Lavender	10	4.25	yes
4	Super Hero Spry Marigold Seeds	Marigold	50	2.95	no
5	Zinnia Zinderella Lilac	Pink	25	3.95	yes
6	Mini Ornamental Mint Seeds	Green	10	3.95	yes
7	Kabloom Light Pink Blast Calibrachoa	Green	10	4.95	yes
8	Calibrachoa Kabloom Coral	Coral	10	4.95	no
9	Fiesta del Sol Mexican Sunflower Seeds	Red	30	3.95	no
10	Cosmos Apricot Lemonade	Yellow	25	3.95	yes
11	Zinderella Purple Zinnia Seeds	Purple	25	3.95	yes
12	Fireball Marigold Seeds	Varies	25	3.95	yes
13	Gerbera Revolution Bicolor Red Lemon	Red	10	8.95	no
14	Paradise Island Calibrachoa Fuseables Seeds	Varies	5	6.95	yes
15	Cheyenne Spirit Coneflower Seeds	Varies	15	7.95	no
16	Leucanthemum Madonna	White	25	4.95	no
17	Zinnia Zinderella Peach	Peach	25	3.95	yes
18	Kabloom Orange Calibrachoa	Orange	10	4.95	yes
19	Fountain Blue Lobelia Seeds	Blue	100	2.50	yes
20	Envy Zinnia Seeds	Green	50	2.95	yes

Use the following data types for your "flower_seeds" columns:

- id - SERIAL
- name - VARCHAR that accepts 300 characters
- main_color - VARCHAR that accepts 100 characters
- seeds_per_packet - INT
- price_per_packet - FLOAT
- in_stock - BOOLEAN

Note: Make sure to save your seed file on your machine so that you can pipe it into your database in the next phase!

Phase 3: Pipe your seed file into your new database

After you've saved your seed file, use the caret and pipe methods to seed your `farm` database with the data from the "edible_seeds" table. (*Note: Make sure you've quit `psql` first with `\q`.*)

There are two ways to seed your database:

Method 1. Seed your database via caret method

```
psql -d [database] -U [username] < [path_to_file/file.sql]
```

Method 2. Seed your database via pipe method

```
cat [path_to_file/file.sql] | psql -d [database] -U [username]
```

Try both of these methods.

If you want to seed using the file again, you need to first drop the tables.

Access the database by:

```
psql -d [database] -U [username]
```

Then drop the tables:

```
DROP TABLE [table];
```

Then you can run the seed file again using any of the two above methods.

Check to make sure your seed file has actually updated the `farm` database

- Log into `psql` again.
- Connect to the `farm` database (syntax: `\c [database]`).
- Make sure your "edible_seeds" table is filled with seeds: `SELECT * FROM «table name»;`
- Make sure your "flower_seeds" table is filled with seeds: `SELECT * FROM «table name»;`

Solving The SQL Menagerie

In our SQL readings, we learned how to write basic SQL queries and incorporate `WHERE` clauses to filter for more specific results. We also learned how to use a `JOIN` operation to get information from multiple tables.

In this project, put your SQL knowledge to the test and show off your querying skills.

We've put together a collection (let's call it a *menagerie*) of SQL problems for you to solve below. Solve them all, and you'll be the master of the menagerie!

Getting started

Clone the starter repository from https://github.com/appacademy-starters/sql-select-exercises-starter.

Project overview

1. In Phase 1, pipe the seed file into a new database.
2. In Phase 2, query the seed tables with basic `SELECT` statements.
3. In Phase 3, query the seed tables using `WHERE` clauses to get more specific rows back.
4. In Phase 4, use a `JOIN` operation to get data from both seed tables.
5. Bonuses! Go beyond what we learned in the readings to deepen your SQL query knowledge.

Phase 1: Pipe in a seed file to create new database tables

We've set up a seed file for you to use in this project called ***cities_and_airports.sql*** that will create two tables: a "cities" table and an "airports" table. These tables show the top 25 most populous U.S. cities and their airports, respectively. Pipe this file into your database, and use these tables for the rest of the project phases.

Go through the following steps:

1. Log into `psql`.
2. Create a new database called "travel".
3. Pipe the *cities_and_airports.sql* seed file into the "travel" database.
4. Check that there's data in both the "cities" and "airports" tables.

Phase 2: Write basic SELECT statements

Retrieve rows from a table using `SELECT` FROM SQL statements.

1. Write a SQL query that returns the city, state, and estimated population in 2018 from the "cities" table.
2. Write a SQL query that returns all of the airport names contained in the "airports" table.

Phase 3: Add WHERE clauses

Select specific rows from a table using WHERE and common operators.

1. Write a SQL query that uses a `WHERE` clause to get the estimated population in 2018 of the city of San Diego.
2. Write a SQL query that uses a `WHERE` clause to get the city, state, and estimated population in 2018 of cities in this list: Phoenix, Jacksonville, Charlotte, Nashville.
3. Write a SQL query that uses a `WHERE` clause to get the cities with an estimated 2018 population between 800,000 and 900,000 people. Show the city, state, and estimated population in 2018 columns.
4. Write a SQL query that uses a `WHERE` clause to get the names of the cities that had an estimated population in 2018 of at least 1 million people (or 1,000,000 people).
5. Write a SQL query to get the city and estimated population in 2018 in number of millions (*i.e. without zeroes at the end: 1 million*), and that uses a `WHERE` clause to return only the cities in Texas.
6. Write a SQL query that uses a `WHERE` clause to get the city, state, and estimated population in 2018 of cities that are NOT in the following states: New York, California, Texas.
7. Write a SQL query that uses a `WHERE` clause with the `LIKE` operator to get the city, state, and estimated population in 2018 of cities that start with the letter "S". (*Note: See the PostgreSQL doc on Pattern Matching for more information.*)
8. Write a SQL query that uses a `WHERE` clause to find the cities with either a land area of over 400 square miles OR a population over 2 million people (or 2,000,000 people). Show the city name, the land area, and the estimated population in 2018.
9. Write a SQL query that uses a `WHERE` clause to find the cities with either a land area of over 400 square miles OR a population over 2 million people (or 2,000,000 people) -- but not the cities that have both. Show the city name, the land area, and the estimated population in 2018.
10. Write a SQL query that uses a `WHERE` clause to find the cities where the population has increased by over 200,000 people from 2010 to 2018. Show the city name, the estimated population in 2018, and the census population in 2010.

Phase 4: Use a JOIN operation

Retrieve rows from multiple tables joining on a foreign key.

The "airports" table has a foreign key called `city_id` that references the `id` column in the "cities" table.

1. Write a SQL query using an INNER JOIN to join data from the "cities" table with data from the "airports" table using the `city_id` foreign key. Show the airport names and city names only.
2. Write a SQL query using an INNER JOIN to join data from the "cities" table with data from the "airports" table to find out how many airports are in New York City using the city name. (*Note: Use the aggregate function COUNT() to count the number of matching rows.*)

Bonuses

1. **Apostrophe:** Write a SQL query to get all three ID codes (*the Federal Aviation Administration (FAA) ID, the International Air Transport Association (IATA) ID, and the International Civil Aviation Organization (ICAO) ID*) from the "airports" table for Chicago O'Hare International Airport. (*Note: You'll need to escape the quotation mark in O'Hare. See How to include a single quote in a SQL query.*)
2. **Formatting Commas:** Refactor Phase 2, Query #1 to turn the INT for estimated population in 2018 into a character string with commas. (*Note: See Data Type Formatting Functions*)
    - Phase 2, Query #1: Write a SQL query that returns the city, state, and estimated population in 2018 from the "cities" table.
3. **Decimals and Rounding:** Refactor Phase 3, Query #5 to turn number of millions from an integer into a decimal rounded to a precision of two decimal places. (*Note: See Numeric Types and the ROUND function.*)

    - Phase 3, Query #5: Write a SQL query to get the city and estimated population in 2018 in number of millions (*i.e. without zeroes at the end: 1 million*), and that uses a `WHERE` clause to return only the cities in Texas.
4. **ORDER BY and LIMIT Clauses:** Refactor Phase 3, Query #10 to return only one city with the biggest population increase from 2010 to 2018. Show the city name, the estimated population in 2018, and the census population in 2010 for that city. (_Note: You'll do the same calculation as before, but instead of comparing it to 200,000, use the ORDER BY Clause with the LIMIT Clause to sort the results and grab only the top result.)
    - Phase 3, Query #10: Write a SQL query that uses a `WHERE` clause to find the cities where the population has increased by over 200,000 people from 2010 to 2018. Show the city name, the estimated population in 2018, and the census population in 2010.