Project Tracker

SQL and Relational Databases

In this exercise, you'll be building a database that stores a list of students and their student projects. There are many ways to do this, but by *modeling* your data appropriately, that data becomes easier to manipulate. The difficulty in using SQL and databases comes not from accessing the data, but from modeling the data in a way that is easy to access.

Go into your project directory and enter the PostgreSQL console.

```
$ psql
```

Now that you're inside the PostgreSQL console, enter the following commands to make a database called ***hackbright*** and connect to your new database:

```
=# CREATE DATABASE hackbright;
CREATE DATABASE
=# \c hackbright
```

At this point, your ***psql*** console should show the database name before =# part of the prompt to indicate you're connected.

Step 1: The *students* Table

You're connected to this shiny new database, but there's nothing in it for you to use yet. For this project tracker, the first thing you'll want to be able to do is store students' real names and GitHub usernames.

Creating a Table

Let's create a ***students*** table to store this information. The table will look something like this:

first_name	*last_name*	*github*
Jane	*Hacker*	*jhacks*
Sarah	*Developer*	*sdevelops*

You can define a table by declaring the names of the columns and what data type can be put into each column. SQL data types largely map directly to standard Python data types. For example, the the SQL ***VARCHAR*** type maps to the Python ***string*** data type.

One quirk of using the ***VARCHAR*** type is that you'll have to explicitly say up front the maximum length of every string you'll ever use. A maximum string length can be hard to predict, but it's harder to expand the field later

to allow bigger strings, so pick a reasonable, conservative default. We'll use the following SQL to create the ***students*** table:

```
CREATE TABLE students (
    first_name VARCHAR(30),
    last_name VARCHAR(30),
    github VARCHAR(30)
    );
```

Enter that definition in your PostgreSQL console now to make the table:

```
=# CREATE TABLE students (
(#     first_name VARCHAR(30),
(#     last_name VARCHAR(30),
(#     github VARCHAR(30)
(#     );
CREATE TABLE
```

You can check that you entered your table schema correctly by asking PostgreSQL to show you the schema. Try it with the ***d*** command now:

```
=# \d students
     Column   |         Type          | Modifiers
  ------------+-----------------------+-----------
   first_name | character varying(30) |
   last_name  | character varying(30) |
   github     | character varying(30) |
```

The output should show the names of your three columns, alongside their data types. Compare this output to the SQL you used to create the table to make sure the information you see in PostgreSQL reflects your specifications.

> **Note: Conventions for Entering SQL**
>
> SQL databases generally don't care about whether table names and field names are capitalized or not, and SQL terms, like ***CREATE TABLE*** and ***VARCHAR***, are always-case insensitive. That said, it's conventional to write SQL as follows:
>
> - SQL terms and types in all uppercase, as in ***CREATE TABLE*** and ***VARCHAR***
> - Field names in all lowercase, as in ***first_name***
> - Table names in all lowercase, as in ***students***
>
> You don't *have* to follow these conventions, but we'll be using it in our documents, as it helps you more easily recognize what something is by how it's written.
>
> Similarly, SQL is easier to read when it's shown with different parts on different lines, like the examples above, but SQL databases don't care whether a command is written this way or all on one line. The command will go into the system in exactly the same way.

So, you could have entered the following SQL to create the ***students*** table:

```
=# create table students(first_name VARCHAR(30),
(# last_name VARCHAR(30), `github VARCHAR(30));
```

This should work, but it's pretty hard to follow, isn't it?

We encourage you to use the conventions shown in our examples, to make your SQL queries easier to for you and anyone else you're working with to read.

Inserting Data

Your project tracker has no students to track at the moment, so let's give the the ***students*** table some records. This process is called *inserting* data into the table.

You can do this with the SQL ***INSERT*** command. Please enter the following command into the PostgreSQL console:

```
=# INSERT INTO students (first_name, last_name, github)
-#     VALUES ('Jane', 'Hacker', 'jhacks');
```

This should have inserted the first row into the ***students*** table. Now, insert a second row for Sarah Developer, with a GitHub username of sdevelops.

Warning: Don't Use Double Quotes!

In Python and JavaScript, you can use single and double quotes interchangeably. However, in SQL, the two quote types are **different**. String values, like 'Jane' and 'Hacker', must use single quotes.

Always use single quotes for strings in SQL.

Note: Shortcut When Providing All Fields

If the values you want to set in your ***INSERT*** statement exactly match the order and number of columns in the table definition, you don't have to specify the order of the columns. SQL will use the order they were defined when the table was created.

The ***Students*** table has ***first_name***, ***last_name***, and ***github*** defined in that order. If you provide that information to your ***INSERT*** statement in the same order, you could write the statement for adding Jane Hacker as:

```
INSERT INTO students VALUES ('Jane', 'Hacker', 'jhacks');
```

Just remember, only use this shortcut if you plan to provide fields to the ***INSERT*** statement in the order the columns appear in your table.

Step 2: Getting Data Out of the Database

Now that you've stored our two students in your database, you can use the SQL you've learned to retrieve the information about those students. Try entering the following command in PostgreSQL:

```
=#  SELECT * FROM students;
```

You can also select individual columns from the table, like this:

```
=#  SELECT last_name FROM students;
```

If you want to see some colums but not all of them, you can select multiple columns, in any order. Here's an example:

```
=#  SELECT github, first_name FROM students;
```

You can also select which rows you want using the ***WHERE*** clauses you've learned about. For example, if you want all of the data for Sarah, you can do the following:

```
=#  SELECT * FROM students WHERE first_name = 'Sarah';
```

You can filter on any value in your table, too. The following command should gives the same search result as the previous:

```
=#  SELECT * FROM students WHERE github = 'sdevelops';
```

Finally, you can combine the two concepts just described to get specific combinations of rows and columns out of your table, like this:

```
=#  SELECT first_name, last_name
-#  FROM students
-#  WHERE github = 'jhacks';
```

When you feel a little more comfortable querying your database for information, move on to the next step.

Step 3: The *projects* Table

Now, the project tracker needs a table to store all the projects that students will do.

One important note here is that this table does not indicate *which* student did *which* project. It is simply a record of all the projects that everyone *can* do.

The table should contain data like this:

title	*description*	*max_grade*
Markov	*Tweets generated from Markov chains*	*50*
Blockly	*Programmatic Logic Puzzle Game*	*100*

Go ahead and create a ***projects*** table with the ***CREATE TABLE*** command as follows:

```
=#  CREATE TABLE projects (
(#     title VARCHAR(30),
(#     description TEXT,
(#     max_grade INTEGER
(#     );
```

This looks pretty similar to the command you used to create the ***students*** table, but it introduces some new SQL data types.

The ***TEXT*** data type is intended for strings that could be very long. This type isn't as small, efficient, or fast as a ***VARCHAR(n)***, but it's often the right choice for long description fields or body-of-document fields, where the length might be unpredictable and long.

The ***INTEGER*** type is for integer numbers.

Once you have a ***projects*** table (you can use the ***d*** command as you did in Step 1 to check that the columns are correct), you can populate it. Add some data for the Markov project with the ***INSERT*** command:

```
=#  INSERT INTO projects (title, description, max_grade)
-#     VALUES ('Markov', 'Tweets generated from Markov chains', 50);
```

Add the information for Blockly to the ***projects*** table, as well.

Now's a great time to get a code review.

Step 3b: Primary Keys and Uniqueness

Let's run one of those insert statements for the ***projects*** table a second time:

```
=#  INSERT INTO projects (title, description, max_grade)
-#    VALUES ('Markov', 'Tweets generated from Markov chains', 50);
```

Select all rows from the table to see all the data in it:

```
=#  SELECT * FROM projects;
```

Oh no! The ***projects*** table has a duplicate entry for the Markov project! We probably didn't want that. Why in the world did you listen to me when I told you to run that insert again?!

No problem, maybe we can just delete that extra project. ***DELETE*** works like ***SELECT***, but without the column names. The ***FROM*** and ***WHERE*** clauses are the same. In fact, it's good practice to run a ***SELECT*** to see what rows your query matches *before* running a ***DELETE*** command.

Therefore, to figure out which row to delete, we just need to write a ***SELECT*** statement to find our duplicate row. How the heck do we do that? The data is the same in both rows. Hmmm...badness.

If only there were a way to uniquely identify each row of a table. It'd also be nice if the database would prevent peoples from duplicating rows in the first place. Luckily, databases can do exactly that with *primary keys*.

If you identify a column in your table as a ***PRIMARY KEY*** column, the database will make sure that any data entered into this column is unique for all the rows of the table.

Numbers make really good primary keys — and you can even let the database handle creating the keys for you. If you add the ***SERIAL*** keyword, the database will use the next available largest integer as the primary key for the table if you leave the primary key out of your ***INSERT*** statement.

The improved ***projects*** table will include a column to store the primary key. The syntax for a primary key column looks like this:

```
id SERIAL PRIMARY KEY
```

If you try to add a ***projects*** table with that key now, however, you'd get an error message that the table already exists, because you already created it earlier in the exercise. Therefore, you'll need to *drop* the ***projects*** table first, using the ***DROP TABLE*** command. Do this now:

```
=#  DROP TABLE projects;
```

Here's the improved SQL to create a ***projects*** table with an integer primary key:

```
=#  CREATE TABLE projects (
(#      id SERIAL PRIMARY KEY,
(#      title VARCHAR(30),
(#      description TEXT,
(#      max_grade INTEGER);
```

Once you've dropped the table, run this improved ***CREATE TABLE*** SQL to create a new ***projects*** table with the autoincrementing ***id*** field.

> **Warning: Be careful!**
>
> ***DROP TABLE*** will delete the table and all your data. Use it with care!

Then, re-run the same ***INSERT*** statements you used before to add projects to the ***projects*** table:

```
=#  INSERT INTO projects (title, description, max_grade)
-#    VALUES ('Markov', 'Tweets generated from Markov chains', 50);
```

```
=#  INSERT INTO projects (title, description, max_grade)
(#    VALUES ('Blockly', 'Programmatic Logic Puzzle Game', 10);
```

These should work just fine. If you select all the information in the table, you should see that each project was given a unique ID, even though you didn't include one.

Make up three additional projects and insert them into your table.

Please stop here and get a code review.

Step 4: Dump/Restore a PostgreSQL Database

Surprises happen. Changes happen, especially when multiple people are working on the same software. As such, the skill of backing up your database (a.k.a., the *database dump*), is very important.

PostgreSQL can generate a text file of all the commands required to re-build the database, and this file can be read and edited by humans. This kind of file, an *SQL dump* file, is useful for backup. In addition, if you want to store your database work on GitHub in order to work with it on multiple computers, you'll need to create an SQL dump file to push to GitHub.

To dump your database, first either quit from PostgreSQL or open a new terminal, because youl'll need to be in the regular console. Then, enter the following command:

```
$ pg_dump hackbright > hackbright.sql
```

This command dumps the SQL to build the database ***hackbright*** into the file ***hackbright.sql***. Now, you can look at that SQL file in a text editor. Open Sublime Text and enter the following to read the SQL:

```
$ subl hackbright.sql
```

If you want to restore your database, re-create it on a new machine, or save edits you made to the SQL file in your database you'll need to re-read that SQL into ***psql***. Follow these steps to do that now:

1. If the database exists already, you'll want to *drop* that database, like this:

   ```
   $ dropdb hackbright
   ```

This command says, “Remove the existing database called ***hackbright***.” Dropping the database is unnecesary if you are creating it for the first time on a new machine.

2. Now, create your database anew:

```
$ createdb hackbright
```

This line says, “Create an empty database called ***hackbright***.” This line is required whether your database previously existed on the machine or not.

3. Once you’ve created a new empty database, read the SQL you dumped into it like this:

```
$ psql hackbright < hackbright.sql
```

In English, this command says, “Run PostgreSQL, and redirect the file ***hackbright.sql*** *in* to that program.”

The **<** operator is like the redirect-output shell operator (**>**), but it works in the other direction: the contents of ***hackbright.sql*** are fed into ***psql***. And since ***hackbright.sql*** contains valid SQL statements, PostgreSQL will execute them, to recreate the database.

Try this out a few times to make sure you understand how to do it, and ask for help if you run into any errors. You’ll need to understand how to do this in order to check your database into Git.

Step 5: Advanced Querying

Now that you have numeric data in your database, you can do more interesting things with the ***SELECT*** statement on your ***projects*** table. For instance, you can do numerical comparisons, like this one:

```
=#  SELECT title, max_grade
-#    FROM projects
-#    WHERE max_grade > 50;
```

Which project or projects should you see in the output? Try this now to see if you were correct.

You can also can compose ***WHERE*** clauses together, joined with the ***OR*** and ***AND*** operators. Enter this example that uses ***AND*** into ***psql*** now:

```
=#  SELECT title, max_grade
-#    FROM projects
-#    WHERE max_grade > 10
-#    AND max_grade < 60;
```

This query selects the title and maximum possible grade from the ***projects*** table where the maximum grade is between 10 and 60.

Let's try a ***WHERE*** clause that uses ***OR*** next:

```
=#  SELECT title, max_grade
-#    FROM projects
-#    WHERE max_grade < 25
-#    OR max_grade > 75;
```

This statement selects all the projects where the maximum grade is less than 25 or more than 75, but not between the two.

Lastly, you can dictate in which order the results are returned to you by adding an ***ORDER*** clause to your ***SELECT*** statement:

```
=#  SELECT *
-#    FROM projects
-#    ORDER BY max_grade;
```

This orders the results by the ***max_grade*** column, in numerical ascending order.

Think about other combinations of information from the ***projects*** table that you might like to look at, and practice some ***SELECT*** statements like those above, with different grade ranges and different row orderings. Feel free to add some more records to the ***projects*** table for more varied output, if you like.

Step 6: Linking the Tables Together

You have two tables, ***students*** and ***projects***, and so far they are completely unrelated. One table is simply a list of students, and the other is a list of projects. You'll need a way to indicate that a student has completed a project and has received a grade.

Create a *grades* Table

To store grades, you'll need a new table; let's call it ***grades***. This table will use data from the ***students*** and ***projects*** tables, and it could hold data like this:

id	*student_github*	*project_title*	*grade*
1	*jhacks*	*Markov*	*10*
2	*jhacks*	*Blockly*	*2*
3	*sdevelops*	*Markov*	*50*
4	*sdevelops*	*Blockly*	*100*

Construct the ***CREATE TABLE*** statement for ***grades*** on your own, using the previous examples as a guide, and create this table in your database now.

The ***student_github*** column in ***grades*** should be the same size and type as the ***github*** column in the ***students*** table. The ***project_title*** should be similarly matched to the ***title*** column in the ***projects*** table.

Insert Grade Records

The data above shows that the student with the GitHub account jhacks has completed the project with the title Markov for a total grade of 10. She has also completed the project entitled Blockly for a measly two points. (She probably chose the Spaceman, and Joel warned everyone that the Panda was way, way cuter.)

The student with the GitHub account sdevelops, on the other hand, is doing much better with her projects, scoring 50 and 100 on each.

The command to insert the first row is written as such:

```
=#  INSERT into grades (student_github, project_title, grade)
-#     VALUES ('jhacks','Markov', '10');
```

Insert this and the remaining rows from the example we showed above into the ***grades*** table, and add some others of your own.

Stop here, high-five your pair, and ask for a code review.

Step 7: Getting Jane's Project Grades

Even though there's no physical link between your tables, there are columns in the ***grades*** table that are meant to correspond to columns in the other two tables. Furthermore, the values in those columns refer to values in the other tables. You can use this characteristic to *join* the data across the columns.

Imagine you want to find the first name, last name, project title, project grade, and maximum grade for that project for a particular student.

To visualize, if you asked about Jane, you'd want:

first_name	*last_name*	*project_title*	*grade*	*max_grade*
Jane	*Hacker*	*Markov*	*10*	*50*
Jane	*Hacker*	*Blockly*	*2*	*100*

You can't currently find all of that information in any one table. The data is actually spread across multiple tables. But you *can* still get all of the information you need at once.

To get this kind of result, we'll be joining the two tables. You may not have heard about joins in lecture, but no need to fret. We'll walk through how to use them.

Building a Query in Parts

Whenever you need to build a complex SQL query, it's helpful to start by creating simpler queries for the individual pieces of information you want. So first, let's build a ***SELECT*** statement for ***first_name*** and ***last_name*** from the ***students*** table. Think of this as **Query 1**:

```
SELECT first_name, last_name
FROM students
WHERE github = 'jhacks';
```

Next, let's select the ***grade*** and ***project_title*** for a student with a particular ***student_github*** value from the ***grades*** table. Think of this as **Query 2**:

```
SELECT project_title, grade
FROM grades
WHERE student_github = 'jhacks';
```

Now we need to select the ***title*** and ***max_grade*** from the ***projects*** table. Think of this as **Query 3**:

```
SELECT title, max_grade
FROM projects;
```

Keep these three queries in mind going forward.

Joining the *students* and *grades* Tables

To mush all that information together, you'll use the ***JOIN*** feature of ***SELECT*** statements. For example, this join will grab information about all students and their grades:

```
SELECT *
FROM students
  JOIN grades ON (students.github = grades.student_github);
```

Try it now, and you should see results similar to the data in the table below.

first_name	*last_name*	*github*	*id*	*student_github*	*project_title*	*grade*
Jane	*Hacker*	*jhacks*	*1*	*jhacks*	*Markov*	*10*
Jane	*Hacker*	*jhacks*	*2*	*jhacks*	*Blockly*	*2*
Sarah	*Develops*	*sdevelops*	*3*	*sdevelops*	*Markov*	*50*
Sarah	*Develops*	*sdevelops*	*4*	*sdevelops*	*Blockly*	*100*

Notice how the output shows the grades for a given student on the same row as their first name.

When you do a ***JOIN***, you'll need to be specific in your query about how to combine the two tables. The ***ON*** statement tells the query what fields will match up. So we're saying: "Hey query, if the github fields match, you can put the rows from the two tables together for our result."

Choosing the Columns You Want

Next, you'll want to limit the columns. As you construct your query this time, you'll need to say which column comes out of which table.

To specify that information, you can use *dot syntax* to identify the table and field, as in ***students.first_name*** and ***grades.project_title***.

This new query is like combining **Query 1** and **Query 2** from above, and it looks like this:

```
SELECT students.first_name, students.last_name, grades.project_title, grades.grade
FROM students
  JOIN grades ON (students.github = grades.student_github);
```

This query should reduce the columns shown to something like this:

first_name	*last_name*	*project_title*	*grade*
Jane	*Hacker*	*Markov*	*10*
Jane	*Hacker*	*Blockly*	*2*
Sarah	*Develops*	*Markov*	*50*
Sarah	*Develops*	*Blockly*	*100*

Adding the *max_grade* Column

Next, you'll need to get the ***max_grade*** out of the ***projects*** table, as in **Query 3**. To do this, you can stack joins on top of each other. In this case, the common data connecting projects and grades is the ***title*** column in the ***projects*** table, and it needs to be joined on the ***project_title*** column in the ***grades*** table.

First, select everything to make sure it all lines up:

```
SELECT *
FROM students
  JOIN grades ON (students.github = grades.student_github)
  JOIN projects ON (grades.project_title = projects.title);
```

Look at the output, and check that you see all columns from the ***students***, ***grades***, and ***projects*** tables.

Filtering for Jane's Information

You can also add a ***WHERE*** clause to show only the lines for the student with the ***github*** column set to jhacks:

```
SELECT *
FROM students
  JOIN grades ON (students.github = grades.student_github)
  JOIN projects ON (grades.project_title = projects.title)
WHERE github = 'jhacks';
```

Now, you should have all available information for just Jane Hacker's projects.

The Final Query

You can get all the data on Jane from this database, but at the start of Step 7, you were only asked for five pieces of information:

- first name
- last name
- project title
- grade
- maximum grade for that project

To achieve the desired output, you'll once again need to filter down to only the columns you want.

For this final step, write a query that selects only the columns that match the example table from the beginning of this step.

When you are done, remember to run ***pg_dump hackbright > hackbright.sql*** and check ***hackbright.sql*** into git along with your other files.

Please stop here and ask for a code review.

If there's time, you can also check out the Further Study <further-study.html>.