Lab 1 : Introduction to SQL
CMSC 362
Marmorstein
Spring 2025
Due: Thursday, Jan. 23rd by 11:59:59pm

The purpose of this lab is to review the SQL commands you should have learned in CMSC 262. In particular, we will go over the commands for creating an SQL table, populating it with data, and then querying it. In the first part of the lab, I will walk you through the creation of a simple table which stores information about missing animals. Then will ask you to create your own table which stores a list of "faculty evaluations". Finally, I will have you populate your database with some example data and execute a few queries against it.

Step 1. Setting Up

I have set up a Postgresql database server on the CMSC lab server. I have created one database for each of you on that server. You also have an account which can access that database. The name of the database, your user name, and your password are all set to your last name (all lowercase). To connect to the server you can type:

```
psql -h postgres -U lastname
```

You will probably be prompted for a password. Type your last name (all lowercase).

This will dump you into a client program (sort of like an SQL interpreter) which lets you type SQL commands to be executed by the server. A better way to execute commands however is to put your SQL statements in a file and tell the server to execute them.

Create a folder named Lab1. In that folder create a file named "lab1.psql". All of your sql code will go in this file.

Comments in SQL begin with two dashes. On the first three lines of your lab1.psql file, add comments for your name, the course number, and “Lab 1”. For example, my file looks like this:

```
-- Robert Marmorstein
-- CMSC 362
-- Lab 1
```

Save your file and quit. At the command prompt, type:

```
psql -h postgres -U lastname -f lab1.psql
```

to execute all of the commands in the file. For now, nothing will happen (the file doesn’t have anything but comments), but this will let us test that you have your connection to the database working properly.

Step 2. Creating a Table

To create a new table in SQL, you use the "CREATE TABLE" command. The syntax looks like this:

```
CREATE TABLE name (   attribute     type,
                      attribute     type,
                      attribute     type,
                      ... );
```

It is considered good style to line up your attributes on separate lines. If you don't use good style, it will still work. However, in this class you will **ALWAYS** line up your attributes so that I can read your code more easily.

The attribute is just a string like "first_name".

The type can be one of several values. The SQL standard specifies that databases must support:

BOOLEAN	: Boolean values (can be True, False, or NULL)
CHAR(N)	: Fixed length strings of exactly **N** characters
VARCHAR(N)	: Variable length strings of up to **N** characters
BIT(N)	: Fixed length strings of exactly **N** binary digits
BIT VARYING(N)	: Variable length strings of up to **N** binary digits
NUMERIC(p, s) or DECIMAL(p,s)	: Fixed point real numbers with **p** total digits of precision and **s** decimal places
INTEGER	: 32-bit signed integers
SMALLINT	: 16-bit small integers
REAL	: Single precision floating point values
FLOAT or DOUBLE PRECISION	: Double precision floating point values
DATE	: Calendar Dates
TIME	: Time of Day
TIMESTAMP	: A date and time combined into a single value
INTERVAL	: An interval of time (can be in microseconds through millennia)
BLOB/CLOB	: Binary/Character Large Objects

Postgres supports all of these, except for BLOB/CLOB (it provides a “BYTEA” type instead).

It also supports some additional types, including arrays, enumerations, geometric types (such as points, lines, line segments, and circles), and many others. Here are some of the interesting ones:

BIGINT	: A 64-bit signed integer
MONEY	: A quantity of some currency (not necessarily dollars)
INET	: A dotted-decimal IP address
TEXT	: An arbitrarily long string
SERIAL	: An alias for “INTEGER” that also creates a sequence object. A sequence is a bit like a counter variable: every time you insert a record into the table, the SERIAL column will automatically be incremented with a new value (unless you hard code the value to something else).
POINT	: An (x,y) pair
BOX	: A pair of points representing the corners of a box.

For a complete list of supported types, see https://www.postgresql.org/docs/current/datatype.html

Let’s practice creating a table. Add the following code to your lab1.psql file:

```
CREATE TABLE lostandfound (  id            SERIAL,
                             animal_type   VARCHAR(20),
                             date_found    DATE,
                             cage          INTEGER,
                             weight        DOUBLE PRECISION);
```

Now run "psql -h postgres -U lastName -f lab1.psql" again.

If you try to run this command twice you will get an error the second time. Why? Because the table already exists!

To solve this problem, you could add "DROP TABLE lostandfound;" to the top of your lab1.psql file directly beneath the comment, but this would create errors if the table doesn't exist. That would be the case if we’d already dropped it for some reason – or if we hadn’t created it yet.

A solution is to use the special keyword IF EXISTS.

Insert the following between your comments and the CREATE TABLE statement.

```
DROP TABLE IF EXISTS lostandfound;
```

Test this by running

psql -h postgres -U lastName -f lab1.psql”

again. If you get a “notice” you can ignore it. A notice is an informational message, but not an error.

Step 2. Populating your database

You have now created a table which has four attributes. It is time to insert some data into the database.

We can populate the database using the INSERT INTO command.

Type:

INSERT INTO lostandfound (animal_type, date_found, cage, weight) VALUES (‘dog’, '2024-12-04', 3, 25.6);

There are a couple of things you should notice about this statement. First, we had to put single quotes around the value for the date and the string but not around the integer or floating point values. Second, we listed ALL of the attributes (other than the id) followed by the word VALUES and then listed the values IN THE SAME ORDER.

In SQL, you can list the attributes in any order you want as long as you make sure that the data values are ordered in the same way.

Notice that we omitted the “id” attribute. Because it is a SERIAL type, PostgreSQL created a “sequence” object for it. Each time we insert into this table (as long as we don’t specify an explicit id) Postgres will automatically assign one from the sequence and increment the sequence to the next value.

Any other attributes we omit would be populated either with default values (if we specified one) or given a NULL value.

Step 3. Modifying the database

Let's modify the table to add a default value to the cage attribute. We can do this using the "ALTER TABLE" command.

Add the following to your lab1.psql file:

```
ALTER TABLE lostandfound ALTER COLUMN cage SET DEFAULT 1;
```

And run the psql command again.

Then add a NEW line:

```
INSERT INTO lostandfound (animal_type, date_found) VALUES
('cat', '2025-01-03'),
('cat', '2025-01-10');
```

As you will see in a minute, this will add two new entries with a cage value of 1 and a **NULL** value for the weight.

(Notice that we can insert multiple rows by enclosing each set of values in parentheses and separating them with commas.

Step 4. A simple query

It doesn't do us a lot of good to put data INTO the database if we can't get it out again. In order to QUERY data from the database we use the "SELECT" command.

We will talk about the SELECT keyword in much more detail later, but for now all you need to know is that:

```
SELECT * FROM lostandfound;
```

will give you a complete printout of the table.

Go ahead and add "SELECT * FROM lostandfound;" to the bottom of your lab1.psql file. Then rerun the psql command.

Step 5. Updating a row

We can also modify the data in the database. Let's set a weight for the cat found on January 3rd.

Add this line to your lab1.psql:

```
UPDATE lostandfound SET weight=8.5 WHERE date_found='2025-01-03';
```

A few things to notice:

1. We use one equal sign to assign in SQL, but ALSO one equal sign to compare.
2. The date needs to be enclosed in single quotes, but the weight is numeric and doesn't need them.

Step 6. A more advanced query

The general syntax of a basic query is:

SELECT columns
FROM tables
WHERE condition;

(We will learn some more advanced queries later in the course)

The SELECT clause tells the database which columns to print (* is a wildcard that tells it to print them all).

The FROM clause tells the database which tables to get those columns from.

By default, the database will print out the required columns for EVERY row of EVERY table in the FROM list. We can limit this by specifying a WHERE clause that contains a boolean condition. The database will apply that condition to each row and only print the rows where the condition is true. This filtering happens BEFORE the columns are selected, but AFTER the tables are joined together.

Let's add a new query that prints out the date found and weight of the cats in the table. We want ONLY the cats, not the dog and we do not want to see the cage number, type, or id.

Add this to lab1.psql:

```
SELECT date_found, weight FROM lostandfound WHERE animal_type='cat';
```

Notice that we can use animal_type in the WHERE clause, even though we did not list it in the SELECT clause.

Step 7. On your own

I've held your hand through the first five steps. Now I want YOU to create and use a database table. The schema for the table is:

evals(instructor_name : string, discipline: string, course_number : string, student_id : string, helpfulness : integer, availability : integer, comment : string);

The instructor name can be just a last name of up to 40 characters. The discipline can be a four letter code like CMSC. The course_number should be a string of about three digits. The student_id is a string of the form L######## (a letter 'L' followed by eight to ten numeric digits). The comment should not have a size limit. Helpfulness and availability should be integers between 1 and 5.

Add code for these three tasks to the bottom of your lab1.psql file:

A. Use CREATE TABLE to set up the relation described above.

B. Populate it with at least 10 rows of data. Try to make these reasonably realistic. You must use at least three different instructors, three different students, and three different courses.

C. Execute a query that will show all of the information in the table.

Submitting

Upload the lab1.psql file directly to the submit page at

`https://marmorstein.org/~robert/submit`

(Do NOT create a tarball of the Lab1 folder – just submit the lab1.psql file directly!)