CSC561 NoSQL Databases
Lab 1, Part 1: The Relational Database Model / PostgreSQL

Relational Databases:

Relational database management systems (RDBMSs) are set-theory-based systems implemented as two-dimensional tables with rows and columns. The canonical means of interacting with an RDBMS is by writing queries in Structured Query Language (SQL). Data values are typed and may be numeric, strings, dates, uninterpreted /user-defined, or other types. The types are enforced by the system. Importantly, tables can join and morph into new, more complex tables, because of their mathematical basis in relational (set) theory.

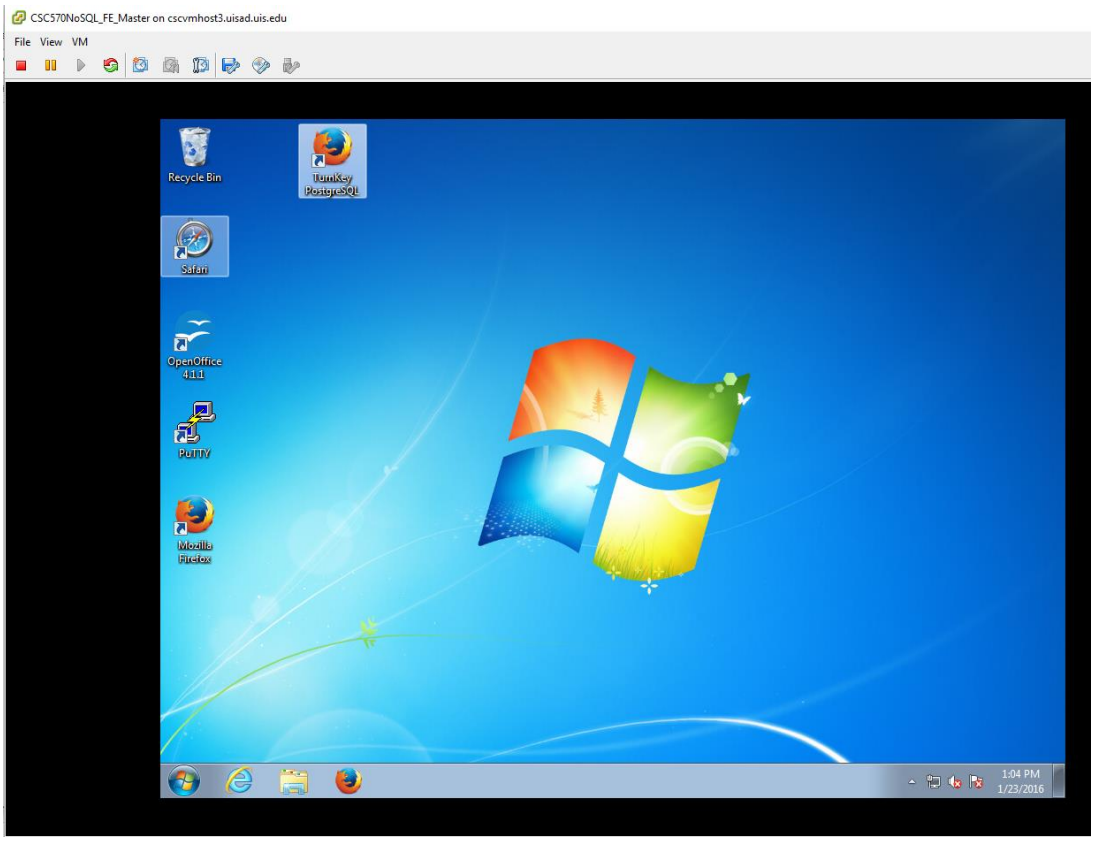
Looking at a Relational Database first will provides a solid point of comparison to the other databases we'll study. It should provide a quick refresher if it has been awhile since you worked with SQL and a good introduction if you are a DB novice. There are some specific advantages to using relational DBs that we want to explore.

Why PostgreSQL:

PostgreSQL is one of the oldest and most robust open-source relational databases. It has adherence to the SQL standard, so it should seem familiar to anyone who has prior experience working with any other relational databases. It provides a solid point of comparison to the other databases we'll work with this semester.

Your Lab Environment:

On your VMware account, you have a Front-end Windows 10 machine and a backend PostgreSQL machine. There is a Firefox Browser shortcut on the Windows 10 desktop that will connect you to your backend.

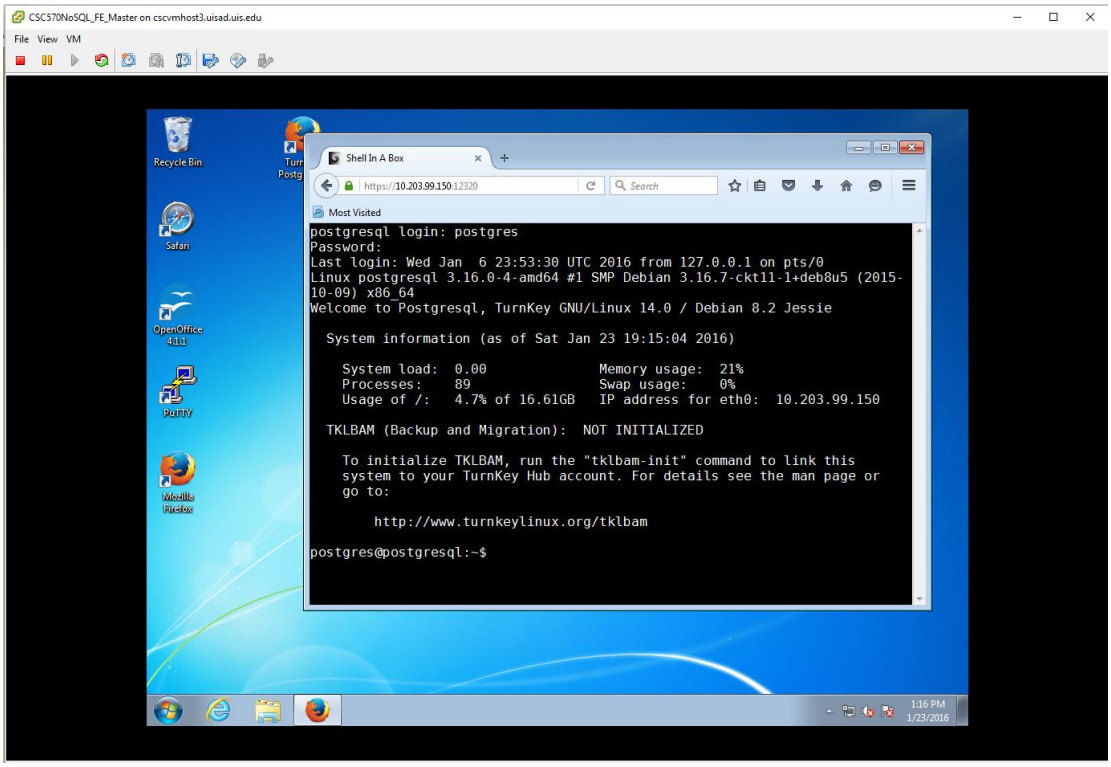


Open the shortcut to your PostreSQL machine and select "Web Shell" interface. The login for your postgres user is:

Postgresql login: postgres

Password: p@ssw0rd

A successful login should show:



Now we'll create our schema to use for this lab by entering the following command:

createdb book

```
postgres@postgresql:~$ createdb book
```

We'll now check to make certain that all of the needed packages are installed in your database workspace by entering:

psql book -c "SELECT '1'::cube;"

If the packages are installed, the command should return 1 row (like below)

```
postgres@postgresql:~$ psql book -c "Select '1'::cube;"
 cube
------
 (1)
(1 row)

postgres@postgresql:~$
```

To drop into the Postgres command-line shell and connect to our created DB enter: psql book

```
postgres@postgresql:~$ psql book
psql (9.4.5)
Type "help" for help.

book=#
```

PostgreSQL prompts with the name of the database followed by a hash mark if you run as an administrator and by dollar sign as a regular user. The shell also comes equipped with the built-in documentation. Typing \h lists information about SQL commands, and \? Helps with psql-specific commands, namely, those that begin with a backslash. You can find usage details about each SQL command by typing \h before the command. For example, if you wanted to know the structure of the CREATE INDEX command, you would enter: \h CREATE INDEX

You should see the following:

```
book-# \h CREATE INDEX
Command:     CREATE INDEX
Description: define a new index
Syntax:
CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] [ name ] ON table_name [ USING method ]
    ( { column_name | ( expression ) } [ COLLATE collation ] [ opclass ] [ ASC | DESC ] [ NUL
LS { FIRST | LAST } ] [, ...] )
    [ WITH ( storage_parameter = value [, ... ] ) ]
    [ TABLESPACE tablespace_name ]
    [ WHERE predicate ]

book-#
```

You can familiarize yourself with this useful tool by reviewing a few common commands, like SELECT or CREATETABLE.

All relational database systems have a SQL language implementation. The SQL implementation for PostgreSQL is called PLPSQL. The documentation for PostgreSQL 9.4 can be found here:
https://www.postgresql.org/docs/9.4/static/plpgsql.html
You can learn more about anything not covered in the lab in this documentation.

Part 1: Relations, CRUD*, and Joins

The tasks we want to complete in this section are to:

1. create schemas
2. populate schemas.
3. query for values
4. join tables

*What is CRUD? CRUD is a useful mnemonic for remembering the basic data management operations: Create, Read, Update, and Delete. These generally correspond to inserting new records (creating), modifying existing records (updating), and removing records you no longer need (deleting). All of the other operations you use a database for (any query) are read operations.

1. Create a Table

Creating a table consists of giving it a name and a list of columns with types and (optional) constraint information. Each table should also nominate a unique identifier column to pinpoint specific rows. That identifier is called a PRIMARY KEY. The SQL to create a **countries** table looks like this:

```
book-# CREATE TABLE countries (
book(# country_code char(2) PRIMARY KEY,
book(# country_name text UNIQUE);
```

This creates a table that will store a set of rows, where each is identified by a two character code and the name is unique. These columns both have constraints. The PRIMARY KEY constrains the **country_code** column to disallow duplicate country codes (for example only one us and one gb may exist). We explicitly gave **country_name** a similar unique constraint (but it is not a primary key).

2. Populate the Table

We can populate the countries table by inserting a few rows:

```
book=# INSERT INTO countries (country_code, country_name)
book-# VALUES ('us', 'United States'), ('mx', 'Mexico'), ('au', 'Australia'),
book-# ('gb', 'United Kingdom'), ('de', 'Germany'), ('ll', 'Loompaland');
```

You can check that the correct data is in the table by reading from it using the select all command: SELECT *

```
book=# SELECT * FROM countries;
 country_code |  country_name
--------------+----------------
 us           | United States
 mx           | Mexico
 au           | Australia
 gb           | United Kingdom
 de           | Germany
 ll           | Loompaland
(6 rows)

book=#
```

Let's delete Loompaland since it isn't a real country. You specify which row to remove by the WHERE clause. The row whose country_code equals 'll' needs to be removed:

```
book=# DELETE FROM countries WHERE country_code = 'll';
```

(You can re-run the "select all" command on the table to make certain it has been removed)

If you want to verify that our constraints are working, you can try to add a duplicate name. The following should give you an ERROR as shown:

```
book=# INSERT INTO countries VALUES ('uk', 'United Kingdom');
ERROR:  duplicate key value violates unique constraint "countries_country_name_key"
DETAIL:  Key (country_name)=(United Kingdom) already exists.
book=#
```

Adding a second table with a foreign key constraint

Now we will create a **cities** table. However, we want to make certain that any **country_code** that exists for a city also exists in our countries table. To do this, we include a REFERENCES keyword. Since the cities table references another table's key, it is known as a *foreign key* constraint.

```
book=# CREATE TABLE cities(
book(# name text NOT NULL,
book(# postal_code varchar(9) CHECK (postal_code <> ''),
book(# country_code char(2) REFERENCES countries,
book(# PRIMARY KEY (country_code, postal_code)
book(# );
CREATE TABLE
book=#
```

In addition to creating a table like before, we also constrained the name in cities by disallowing NULL values. We constrained postal_code by checking that no values are empty strings (<> means not equal). Furthermore, since a PRIMARY KEY uniquely identifies a row, we created a compound key: country_code + postal_code. Together, they uniquely define a row.

Notice that Postgres also has a rich set of datatypes. The above instruction uses three different string representations: text (a string of any length), varchar(9) (a string of variable length up to nine characters), and char(2) (a string of exactly two characters).

Let's verify our foreign key constraint by trying to add a city from a country that doesn't exist in our **countries** table:

```
book=# INSERT INTO cities VALUES ('Toronto', 'M4C1B5', 'ca');
ERROR:  insert or update on table "cities" violates foreign key constraint "cities_country_co
de_fkey"
DETAIL:  Key (country_code)=(ca) is not present in table "countries".
book=#
```

We got an error because 'ca', Canada, is not in our **countries** table, so we have verified that our constraint works. Working constraints like these are what maintain referential integrity within relational databases. Considerations like this are an important part of correctly creating the data model for a database.

3. Populating our second table and updating information

Insert a US city into our **cities** table:

```
book=# INSERT INTO cities VALUES ('Portland', '87200', 'us');
INSERT 0 1
```

This successfully inserts a row. But what if we had made an error in our data? Rather than deleting the row, we can use the UPDATE command. Let's update the postal code for Portland from the incorrect value of 87200 to its correct value of 97205:

```
book=# UPDATE cities
book-# SET postal_code = '97205'
book-# WHERE name = 'Portland';
UPDATE 1
book=#
```

4. Table joins

All databases we'll study perform CRUD operations as well. What sets relational databases like PostgreSQL apart is their ability to **JOIN** tables together when reading them. Joining, in essence, is an operation taking two separate tables and combining them in some way to return a single table.

INNER JOINS: The basic form of a join is the inner join. In the simplest form, you specify two columns (one from each table) to match by, using the ON keyword. Let's create a **JOIN** on **country_code** so that we can get a **country_name** for each city:

```
book=# SELECT cities.*, country_name
FROM cities INNER JOIN countries
ON cities.country_code = countries.country_code;
```

This should return a table sharing all columns' values of the **cities** table plus the matching **country_name** value from the **countries** table:

```
   name   | postal_code | country_code | country_name
----------+-------------+--------------+---------------
 Portland | 97205       | us           | United States
(1 row)

book=#
```

We can also join a table like cities that has a compound primary key. To test a **compound join**, let's create a new table that stores a list of venues.

A venue exists in both a postal code and a specific country. The foreign key must be two columns that reference both cities primary key columns. (MATCH FULL is a constraint that ensures either both values exist or both are NULL.)

```
book=# CREATE TABLE venues (
book(# venue_id SERIAL PRIMARY KEY,
book(# name varchar(255),
book(# street_address text,
book(# type char(7) CHECK ( type in ('public','private') ) DEFAULT 'public',
book(# postal_code varchar(9),
book(# country_code char(2),
book(# FOREIGN KEY (country_code, postal_code)
book(# REFERENCES cities(country_code, postal_code) MATCH FULL
book(# );
CREATE TABLE
book=#
```

This **venue_id** column is a common primary key setup: automatically incremented integers (1, 2, 3, 4, and so on…) for each row insertion. We make this identifier using the SERIAL keyword (MySQL has a similar construct called AUTO_INCREMENT). So to add a row to this new table, enter:

```
book=# INSERT INTO venues (name, postal_code, country_code)
book-# VALUES ('Crystal Ballroom', '97205', 'us');
INSERT 0 1
book=#
```

Although we did not set a venue_id value, creating the row populated it.

Joining the **venues** table with the **cities** table requires both foreign key columns. To save on typing, we can *alias* the table names by following the real table name directly with an alias, with an optional AS between (for example, venues v or venues AS v):

```
book=# SELECT v.venue_id, v.name, c.name
book-# FROM venues v INNER JOIN cities c
book-#  on v.postal_code=c.postal_code AND v.country_code=c.country_code;
```

Which should return:

```
 venue_id |       name       |   name
----------+------------------+----------
        1 | Crystal Ballroom | Portland
(1 row)
```

OUTER JOINS: Outer joins are a way of merging two tables when the results of one table must always be returned, whether or not any matching column values exist on the other table.

5. Obtaining the table id when you insert a new record

There is a nice feature in PostgreSQL that lets us request that PostgreSQL return columns after insertion by ending the query with a RETURNING statement.

INSERT INTO venues (name, postal_code, country_code)
VALUES ('Voodoo Donuts', '97205', 'us') RETURNING venue_id;

This should return a new venue_id without having to issue an additional query:

```
book=# INSERT INTO venues (name, postal_code, country_code)
book-# VALUES ('Voodoo Donuts','97205','us') RETURNING venue_id;
 venue_id
----------
        2
(1 row)

INSERT 0 1
book=#
```

****NOTE ****
To Quit PostgreSQL, enter: \q

CONTINUE TO NEXT PAGE FOR WORK TO SUBMIT

6. *Work on your own to submit*

Now that we've gone through the basics, here are some tasks for you to accomplish on your own. **Everything you write for this lab should be written sequentially in an .SQL text file named lab1.sql saved in the lab1 folder in your GitHub repository. I will not grade your work until after Monday night and will grade the most recent submission. This allows you to test your code as often as needed until it runs as intended.**

Everything up until now has been practice using your VM, which has given you an opportunity to become familiar with new skills in a hands-on environment. At this point, you will be applying those skills on your own. For the steps below, you may continue using your VM to test your code before pushing it to GitHub. Or you may test your code on the continuous integration server and continuing revising/testing until it works. All labs in this course will work in the same way – you will practice on your VM first. Then you may choose to do the labs on your VM to test them before pushing to GitHub or you may do the labs directly on the continuous integration server.

In this lab, we will create tables for an equipment checkout system, typical to what a university help desk would offer. The user comes to the help desk to check out equipment. The status and inventory tables would be pre-populated by the help desk staff. When the user checks out a piece of equipment, a record is entered into the transactions table.

1. If you are working on your VM, create a new database named lab1 to use when completing the following steps. When you submit your code to GitHub, this step is not necessary because the Drone continuous integration server does it automatically.

2. Create a new table named **users**

   It should have the following columns:

   - SERIAL PRIMARY KEY integer id
   - first_name
   - last_name
   - email (not null)
   - password (not null)
   - created_at (of type timestamp)
   - updated_at (of type timestamp)

2. Create a new table named **status**

   It should have the following columns:

   - SERIAL PRIMARY KEY integer id
   - Description (not null)
   - created_at (of type timestamp)
   - updated_at (of type timestamp)

3. Create a new table named **inventory**

   It should have the following columns:

   - SERIAL PRIMARY KEY integer id

- status_id (foreign key constraint by referencing the status table)
- description (not null)
- created_at (of type timestamp)
- updated_at (of type timestamp)

4. Create a new table named **transactions**

   It should have the following columns:

   - SERIAL PRIMARY KEY id
   - user_id (foreign key constraint by referencing the user table)
   - inventory_id (foreign key constraint by referencing the inventory table)
   - checkout_time (of type timestamp) | (not null)
   - scheduled_checkin_time (of type timestamp)
   - actual_checkin_time (of type timestamp)
   - created_at (of type timestamp)
   - updated_at (of type timestamp)

Note: timestamps are a string like 2022-02-15 17:30

5. Insert 5 users into the users table. The fields are self-explanatory.
6. Insert 5 records into the status table. The description field should be: Available, Checked out, Overdue, Unavailable, Under Repair. The other fields are self-explanatory.
7. Insert 5 records into the inventory table. The description field should be: Laptop1, Laptop2, Webcam1, TV1, Microphone1. The other fields are self-explanatory.
8. Insert 3 records into the transactions table. Then update the status of these three inventory items in the inventory table to Checked out. The fields are self-explanatory, keeping in mind the foreign key constraints. Two of the transactions should be for the user in the users table with id = 1. Two of the transactions should have a scheduled_checkin_time after May 31 2022.
9. Alter the users table to add a column for signed_agreement (Boolean column that defaults to false).
10. Write a query that returns a list of all the equipment and its scheduled_checkin_time that is checked out ordered by scheduled_checkin_time in descending order.
11. Write a query that returns a list of all equipment due after May 31 2022.
12. Write a query that returns a count of the number of items with a status of Checked out by user_id 1.

NOTE: You will need to look in the documentation for PostgreSQL 9.4 (https://www.postgresql.org/docs/9.4/static/plpgsql.html) to complete the tasks in the labs that are not covered in the examples. This is good practice for researching and troubleshooting that you will need in your careers.