# CSC570 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 withrows and columns. The canonical means of interacting with an RDBMS is bywriting queries in Structured Query Language (SQL). Data values are typedand may be numeric, strings, dates, uninterpreted /user-defined, or other types. Thetypes are enforced by the system. Importantly, tables can join and morphinto 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 7 machine and a Beckend PostgreSQL machine. There is a Firefox Browser shortcut on the Windows 7 desktop that will connect you to your backend.

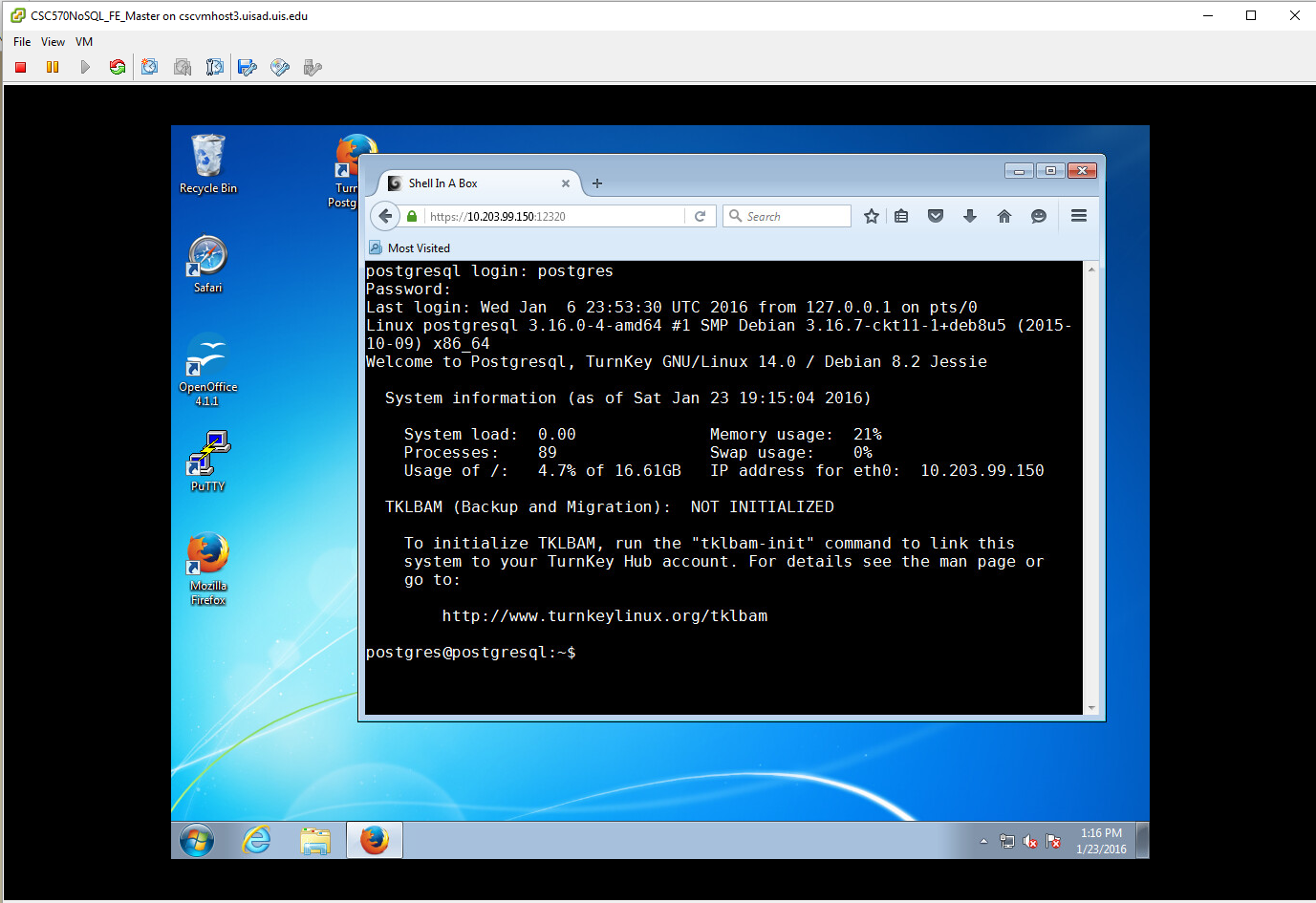


Open the shortcut to your PostreSQL machine and select “Web Shell” interface. The login for your postgres user is:

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:

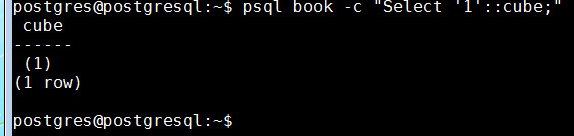
created book



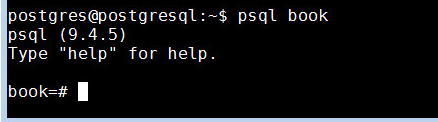
We’ll now check to make certain that all of the needed packages are installed in your database workspace by entering:

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

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

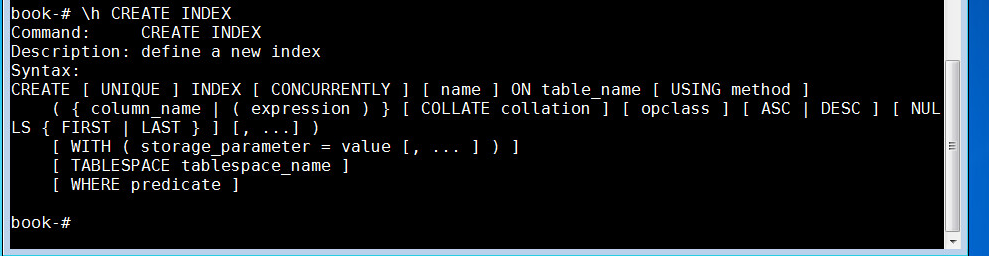


To drop into the Postgres command-line shell and connect to our created DB enter: psql book



PostgreSQL prompts with the name of the database followed by a hash markif 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:



You can familiarize yourself with this useful tool by looking over (or brushing up on) a few common

commands, like SELECT or CREATETABLE .

#### Part 1: Relations, CRUD\*, and Joins

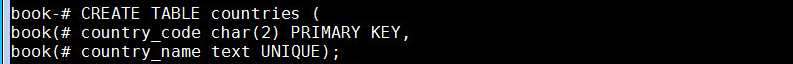
The tasks we want to complete in this section are to:

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:



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).

1. Populate the Table

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



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

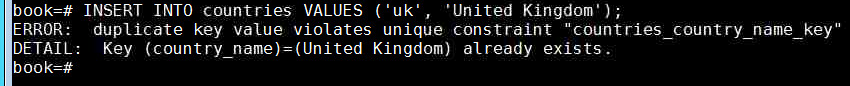


Let’s delete Loompaland since it isn’t a real country. You specify which row to remove by the WHERE clause. The row whose country\_code equals ‘ll’ needs to be removed:



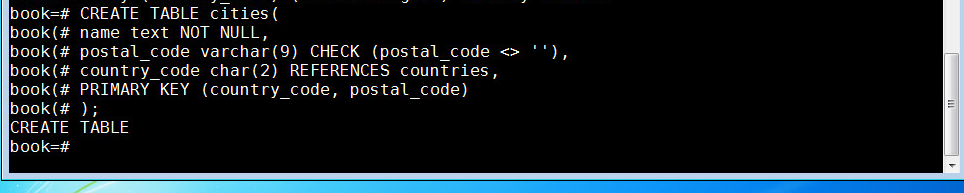
(You can re-run the “select all” command on the table to make certain it has been removed)

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:



1. Adding a second table with a foreign key constraint

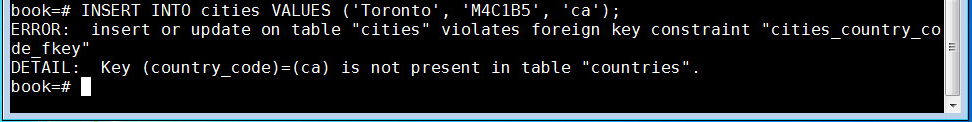
No we will create a **cities** table. However, we want to make certain that any **country\_code** that exists for a city also exists in our countries table. To do this, we include a REFERENCES keyword. Since the cities table references another table’s key, it is known as a *foreign key* constraint.



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:



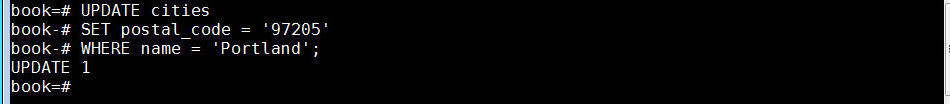
We got an error because ‘ca’, Canada, is not in our **countries** table, so we have verified that our constraint works. Working constraints like these are what maintain referential integrity within relational databases. Considerations like this are an important part of correctly creating the data model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

1. Populating our second table and updating information

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



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:



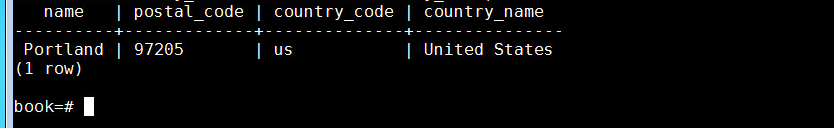
1. Table joins

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:

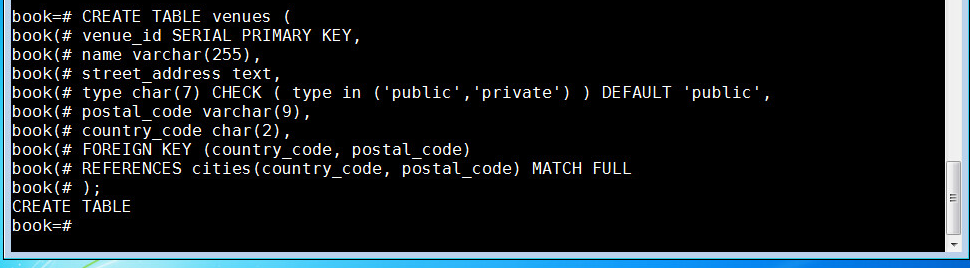


This should return a table sharing all columns’ values of the **cities** table plus the matching **country\_name** value from the **countries** table:

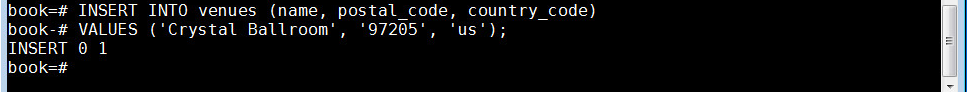


We can also join a table like cities that has a compound primary key. To test a **compound join**, let’s create a new table that stores a list of venues.

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.)



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:

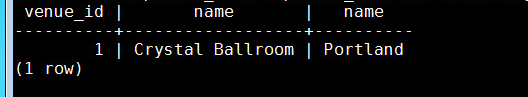


Although we did not set a venue\_id value, creating the row populated it.

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):



Which should return:



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.

\*\*NOTE \*\*

To Quit PostgreSQL, enter: \q

Shutdown your back end and front-end machines when you are finished with the Lab.

**CONTINUE TO NEXT PAGE FOR WORK TO SUBMIT**

1. 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 and submit to blackboard. Take screen shots of your VM output (like in the previous lab steps)and insert these into this word document .

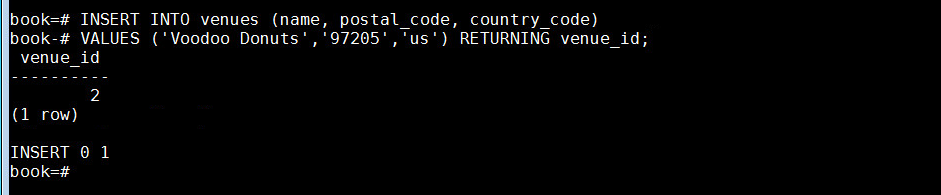
REV 1: **\*\*\*\*STEP 0.5\*\*\*\***

**Add an additional venue to the venue table. 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:**

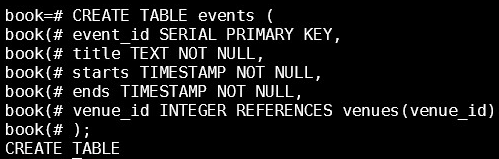


1. Create a new table named **events**

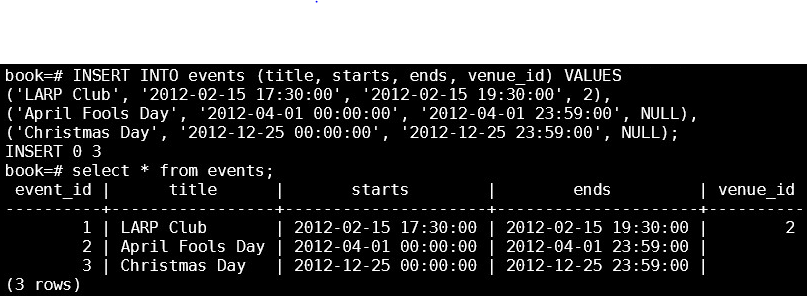
It should have the following columns:

* + - a SERIAL integer event\_id,
    - a title,
    - starts (of type timestamp)
    - ends (of type timestamp),
    - a venue\_id (foreign key that references venues).

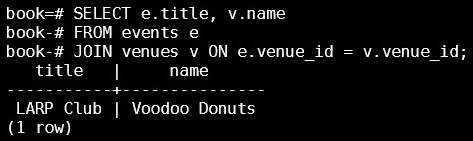
**timestamps** are a string like 2012-02-15 17:30

****

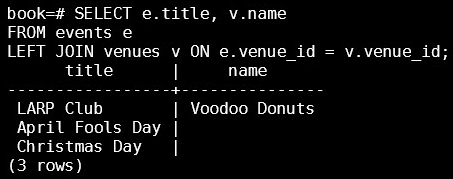
1. Insert the following values: 



1. Write a query that returns an event title and venue name as an inner join. (include your query and what this query returns in your screenshot).



1. Write an outer join query that returns all of the events, whether or not they have a venue. You will need to use a LEFT OUTER JOIN. (include your query and what this query returns in your screenshot).



1. Write a query that finds the country\_name of the LARP Club event. (include your query and what this query returns in your screenshot).

