Relational Databases and SQL, Part II
Statistics 650/750
Week 9 Thursday

Christopher Genovese and Alex Reinhart

26 Oct 2017

Announcements

- SQL Server instructions
- Challenges: Right vs. Fast, Revisions
- Tests: you are doing the work already
- Generalizability: example in autocompete
- Notes and code for today in documents

Commands and Files

Update the `documents` repository from github. There are several files in `ClassFiles/week9` that you should copy into a working directory for this class:

- commands-R.sql
- commands-R.r
- commands-R.py
- instructions
- events.csv
- events.sql

The last of these is a text file containing sql commands that you can copy and paste into the prompt to save typing. Of course, typing the commands is fine too and is not a bad way to get a feel for how the commands work.

Plan

Today

- CRUD Review and Activity Continued
- Joins and Foreign Keys
- Using RDBs Programatically from Your Favorite Programming Language

Later and Appendix

- Schema Design
- A Few Advanced Maneuvers
- A Quick View of Other Database Models

PostgreSQL Primer

Connecting (see also instructions file)

1. For Mac and Linux users, open a terminal. Windows users start git-bash (or equivalent).
2. Log into the new department database server, sculptor.stat.cmu.edu, via ssh using the username and password Carl sent you in email.

   ```
   ssh yourusername@sculptor.stat.cmu.edu
   ```

   You'll be asked for your password. Type it (the letters won't show up), then hit Enter and you should be logged in. If it asks "are you sure you want to continue connecting?", say yes. (You should probably change your password by tying `passwd` at the shell prompt and entering a new password.)
3. At the shell prompt, type

   ```
   psql
   ```

   which will start an interactive postgresql REPL. You should now see a prompt like 'yourusername# '.
4. Work in postgres. To quit psql, type `\q` at the prompt.
5. Mac users with homebrew installed might want to install postgresql locally instead. At the shell prompt, do

   ```
   brew install postgresql
   initdb -D /usr/local/var/postgres
   pg_ctl -D /usr/local/var/postgres start
   createdb NAME
   ```

   where NAME is your username (the word after /Users in your home directory path), for simplicity.

1. Windows users may be able to do this through git-bash. If not, you will need to download PuTTY, a SSH client for Windows. Download `PuTTY.exe` from here:

   `http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html`

   PuTTY users can log in this through the graphical interface, connecting to pg.stat.cmu.edu and using your username and password; you should see a command line prompt once you successfully logged in.

Getting Help

Type '\?' at the prompt to get a list of meta-commands (these are system, not SQL commands).
A few of these are quite common:

- '`\h`' provides help on an SQL command or lists available commands
- '`\d`' list or describe tables, views, and sequences
- '`\l`' lists databases
- '`\c`' connect to a different database
- '`\i`' read input from a file (like source)
- '`\o`' send query output to a file or pipoe
- '\!' execute a shell command
- '`\cd`' change directory
- '`\copy`' copy data into a table
- '`\q`' quit psql

Entering SQL Statements

SQL consists of a sequence of *statements*.
Each statement is built around a specific command, with a variety of modifiers and optional clauses.
SQL statements can span several lines, and all SQL statements end in a semi-colon (;).
Keep in mind: strings are delimited by single quotes 'like this', *not* double quotes "like this".
SQL comments are lines starting with `--`.
To get help:

- You can get brief help on any SQL command with `\h <command>`.
- You can get detailed and helpful information on any aspect of postgres through the online documentation.
- The stat server is running version 9.2, that that will be updated if needed.

A Simple Example

Try the following (or copy it from the given file).

```
1 create table products (
2       product_id SERIAL PRIMARY KEY,
3       name text,
4       price numeric CHECK (price > 0),
5       sale_price numeric CHECK (sale_price > 0),
6       CHECK (price > sale_price)
7 );
```

Then type `\d` at the prompt. You should see the table.
Next, we will enter some data.

```
insert into products (name, price, sale_price) values ('furby', 100, 95);
insert into products (name, price, sale_price)
       values ('frozen lunchbox', 10, 8),
              ('uss enterprise', 12, 11),
              ('spock action figure', 8, 7),
              ('slime', 1, 0.50);
```

Do the following, one at a time.

```
select * from products;
select name, price from products;
select name as product, price as howmuch from products;
```

Discussion...

Managing Tables

Command summary:

- `create table NAME (attribute1 type1, attribute2 type2, ...);`
- `alter table NAME ALTERATION;`

  Common ATERATIONS:

  - Rename column: `rename OLD_NAME to NEW_NAME`
  - Add column: `add NAME TYPE [CONSTRAINTS]`
  - Drop column: `drop NAME`
  - Alter column: `alter NAME set CONFIG`

- `drop table NAME;`

Examples: Creating Tables

```
create table star (
       id SERIAL PRIMARY KEY,
       ra  numeric,
       dec numeric,
       epoch date,
       magnitude numeric
);
```

```
create table products (
       product_id SERIAL PRIMARY KEY,
       name text,
       price numeric CHECK (price > 0),
       sale_price numeric CHECK (sale_price > 0),
```

```
6       CHECK (price > sale_price)
7 );
```

```
1 create table products (
2       product_id SERIAL PRIMARY KEY,
3       label text UNIQUE NOT NULL CHECK (char_length(label) > 0),
4       price numeric CHECK (price >= 0),
5       discount numeric DEFAULT 0.0 CHECK (discount >= 0),
6       CHECK (price > discount)
7 );
```

Examples: Altering Tables

A few examples using the most recent definition of `products` above:

- Change a column name

```
1 alter table products rename product_id to id;
```

- Let's add a `brand_name` column.

```
1 alter table products add brand_name text DEFAULT 'generic' NOT NULL;
```

- Let's drop the `discount` column

```
1 alter table products drop discount;
```

- Let's set a default value for `brand_name`.

```
1 alter table products alter brand_name SET DEFAULT 'generic';
```

Example: Dropping Tables

```
1 drop table products;
```

CRUD Review/Summary

- `INSERT` – populate *new* rows of a table with data

```
INSERT INTO <tablename> (<column1>, ..., <columnk>)
       VALUES (<value1>, ..., <valuek>)
       RETURNING <expression|*>;
```

More than one tuple can be given as values, each for a successive row. The `RETURNING` clause is optional, values can be set to `DEFAULT` to specify the default value.

- `SELECT` – generate values from data in a table in table format

  SELECT is a powerful command for generating data. It's most common use is to read (and possibly transform) data in one or more tables, but it can be used in other ways as well.

  It has many forms, but a common way to read data from a table looks like

  ```
  SELECT expressions FROM source WHERE conditions;
  ```

- `UPDATE` – change the values in selected existing rows

  ```
  UPDATE table
      SET col1 = expression1,
          col2 = expression2,
          ...
      WHERE condition;
  ```

  This can have an optional `RETURNING` clause like `INSERT`.

- `DELETE` – drop rows from the table

  ```
  DELETE FROM table WHERE condition;
  ```

  The `WHERE` clause is optional, but without it, you will delete all the table's rows.

CRUD Examples

To start, let's create a table.

```
create table events (
       id SERIAL PRIMARY KEY,
       moment timestamp DEFAULT 'now',
       persona integer NOT NULL,
       element integer NOT NULL,
       score integer NOT NULL DEFAULT 0 CHECK (score >= 0 and score <= 1000),
       hints integer NOT NULL DEFAULT 0 CHECK (hints >= 0),
       latency real,
       answer text,
       feedback text
);
```

Note: Later on, `persona` and `element` will be foreign keys, but for now, they will just be arbitrary integers.

Then, copy data from `events.csv` into the events table:

```
\COPY events FROM 'events.csv'
       WITH DELIMITER ',';
SELECT setval('events_id_seq', 1001, false);
```

You should replace the first string by the correct path to the `events.csv` file on your computer.

```
insert into events (persona, element, score, answer, feedback)
       values (1211, 29353, 824, 'C', 'How do the mean and median differ?');
insert into events (persona, element, score, answer, feedback)
       values (1207, 29426, 1000, 'A', 'You got it!')
       RETURNING id;
insert into events (persona, element, score, answer, feedback)
       values (1117, 29433,  842, 'C', 'Try simplifying earlier.'),
              (1199, 29435,    0, 'B', 'Your answer was blank'),
              (1207, 29413, 1000, 'C', 'You got it!'),
              (1207, 29359,  200, 'A', 'A square cannot be negative')
       RETURNING *;
```

Next, try inserting a few valid rows giving latencies but not id or feedback. Find the value of the id's so inserted.

Query all rows and columns of a table:

```
select * from events;
```

The `*` is a shorthand for "all columns."

`SELECT` can generate data with direct expressions or by transforming data from a table. The source in the SELECT statement can be expressions as well as column names. And we can use `as` clauses to name (or rename) the results.

```
select 1 as one;
select ceiling(10*random()) as r;
select 1 from generate_series(1,10) as ones;
select min(r), avg(r) as mean, max(r) from
       (select random() as r from generate_series(1,10000)) as _;
select timestamp '2015-01-22 08:00:00' + random() * interval '64 days'
       as w from generate_series(1,10);
```

The fourth case uses a selct (called a sub-select) to create a virtual table to select from.

A `WHERE` clause constrains which rows are included:

```
select * from events where id > 20 and id < 40;
```

Any logical expression is allowed in the WHERE clause, including referencing column names.

As we will see more next time, we can also order the output using the `ORDER BY` clause and group rows for aggregation using the `GROUP BY` clause values over groups.

```
select score, element from events
    where persona = 1202 order by element, score;
select count(answer) from events where answer = 'A';
select element, count(answer) as numAs
```

```
        from events where answer = 'A'
        group by element
        order by numAs;
select persona, avg(score) as mean_score
        from events
        group by persona
        order by mean_score;
```

Sub-selects can also be used to insert into a table, using the same names.

```
-- table foo has columns a, b, c of types numeric, integer, and text

insert into foo (select random() as a, generate_series(100,105) as b, 'blah' as c);
```

A few SELECT's to consider (see below for answers):

1. List all event ids for events taking place after 20 March 2015 at 8am. (Hint: `>` and `<` should work as you hope.)
2. List all ids, persona, score where a score > 900 occurred.
3. List all persona (sorted numerically) who score > 900. Can you eliminate duplicates here? (Hint: Consider `SELECT DISTINCT`)
4. Can you guess how to list all persona whose average score > 600. You will need to do a `GROUP BY` as above. (Hint: use `HAVING` instead of `WHERE` for the aggregate condition.)
5. Produce a table showing how many times each instructional element was practiced.

```
select id from events where moment > timestamp '2015-03-20 08:00:00';
select id, persona, score from events where score > 900;
select distinct persona from events where score > 900 order by persona;
select persona from events group by persona having avg(score) > 600;
select element, count(element) from events group by element order by element;
```

Create another table for the next examples:

```
create table gems (label text DEFAULT '',
                   facets integer DEFAULT 0,
                   price money);

insert into gems (select '', ceiling(20*random()+1), money '1.00' from generate_series(1,20) as k);

update gems set label = ('{thin,quality,wow}'::text[])[ceil(random()*3)];

update gems set label = 'thin'
        where facets < 10;
update gems set label = 'quality',
                price = 25.00 + cast(10*random() as numeric)
        where facets >= 10 and facets < 20;
```

```
update gems set label = 'wow', price = money '100.00'
       where facets >= 20;

select * from gems;
```

How to do the following?

1. Update events with id $>$ 1000 to set latencies where they are missing. (Consider `select id from events where latency is null;` to find them.)
2. Set answers for id $>$ 1000 to a random letter A through D.
3. Update the scores to subtract 50 points for every hint taken when id $>$ 1000. Check before and after to make sure it worked.

```
delete from gems where facets < 5;
delete from events where id > 1000 and answer = 'B';
```

Try to delete a few selected rows in one of your existing tables. (Remember: you can do `\d` at the prompt to check the table list.)

Activity

Here, we will do some brief practice with CRUD operations by generating a table of random data and playing with it. Start where you left off.

1. Create a table `rdata` with five columns: one `integer` column `id`, two `text` columns `a` and `b`, one `date` `moment`, and one `numeric` column `x`.
2. Use a `SELECT` command with the `generate_series` function to display the sequence from 1 to 100.
3. Use a `SELECT` command with the `random()` function converted to `text` (via `random()::text`) and the `md5` function to create a random text string.
4. Use a `SELECT` command to choose a random element from a fixed array of strings. A fixed text array can be obtained with `('{X,Y,Z}'::text[])` and then indexed using the `ceil` (ceiling) and `random` functions to make a selection. (FYI, `('{X,Y,Z}'::text[])[1]` would give 'X'.) (SQL is 1-indexed.)
5. `SELECT` a random date in 2017. You can do this by adding an integer to `date '2017-01-01'`. For instance, try

   ```
   select date '2017-01-01' + 7 as random_date;
   ```

   For a non-integer type, append `::integer` to convert it to an integer.
6. Use `INSERT` to populate the `rdata` table with 101 rows, where the `id` goes from 1 to 100, `a` is random text, `b` is random choice from a set of strings (at least three in size), `moment` contains random days in 2017, and `x` contains random real numbers in some range.
7. Use `SELECT` to display rows of the table for which `b` is equal to a particular choice.

8. Use `SELECT` with either the `~*` or `ilike` operators to display rows for which `a` matches a specific pattern, e.g.,

```
select * from rdata where a ~* '[0-9][0-9][a-c]a';
```

9. Use `SELECT` with the `overlaps` operator on dates to find all rows with `moment` in the month of November.

10. Use `UPDATE` to set the value of `b` to a fixed choice for all rows that are divisible by 3 and 5.

11. Use `DELETE` to remove all rows for which `id` is even and greater than 2. (Hint: `%` is the mod operator.)

12. Use a few more `DELETE`'s (four more should do it) to remove all rows where `id` is not prime.

Joins and Foreign Keys

As we will see shortly, principles of good database design tell us that tables represent distinct entities with a single authoritative copy of relevant data. This is the DRY principle in action, in this case eliminating *data redundancy*.

An example of this in the `events` table are the `persona` and `element` columns, which point to information about students and components of the learning environment. We do **not** repeat the student's information each time we refer to that student. Instead, we use a **link** to the student that points into a separate `Personae` table.

But if our databases are to stay DRY in this way, we need two things:

1. A way to define links between tables (and thus define *relationships* between the corresponding entities).

2. An efficient way to combine information across these links.

The former is suppled by foreign keys and the latter by the operations known as joins. We will tackle both in turn.

Foreign Keys

A **foreign key** is a field (or collection of fields) in one table that *uniquely* specifies a row in another table. We specify **foreign keys** in Postgresql using the `REFERENCES` keyword when we define a column or table. A foreign key that references another table must be the value of a unique key in that table, though it is most common to reference a *primary key*.

Example:

```
create table countries (
       country_code char(2) PRIMARY KEY,
       country_name text UNIQUE
);
insert into countries
  values ('us', 'United States'), ('mx', 'Mexico'), ('au', 'Australia'),
         ('gb', 'Great Britain'), ('de', 'Germany'), ('ol', 'OompaLoompaland');
select * from countries;
delete from countries where country_code = 'ol';

create table cities (
```

```
        name text NOT NULL,
        postal_code varchar(9) CHECK (postal_code <> ''),
        country_code char(2) REFERENCES countries,
        PRIMARY KEY (country_code, postal_code)
);
```

Foreign keys can also be added (and altered) as *table constraints* that look like `FOREIGN KEY (<key>) references <table>`.

Now try this

```
insert into cities values ('Toronto', 'M4C185', 'ca'), ('Portland', '87200', 'us');
```

Notice that the insertion did not work – and the entire transaction was rolled back – because the implicit foreign key constraint was violated. There was no row with country code 'ca'.

So let's fix it. Try it!

```
insert into countries values ('ca', 'Canada');
insert into cities values ('Toronto', 'M4C185', 'ca'), ('Portland', '87200', 'us');
update cities set postal_code = '97205' where name = 'Portland';
```

Joins

Suppose we want to display features of an event with the name and course of the student who generated it. If we've kept to DRY design and used a foreign key for the `persona` column, this seems inconvenient.

That is the purpose of a **join**. For instance, we can write:

```
select personae.lastname, personae.firstname, score, moment
       from events
       join personae on persona = personae.id
       where moment > timestamp '2015-03-26 08:00:00'
       order by moment;
```

Joins incorporate additional tables into a select. This is done by appending to the `from` clause:

`from <table> join <table> on <condition> ...`

where the `on` condition specifies which rows of the different tables are included. And within the select, we can disambiguate columns by referring them to by `<table>.<column>`. Look at the example above with this in mind.

We will start by seeing what joins mean in a simple case.

```
create table A (id SERIAL PRIMARY KEY, name text);
insert into A (name)
       values ('Pirate'),
              ('Monkey'),
              ('Ninja'),
              ('Flying Spaghetti Monster');

create table B (id SERIAL PRIMARY KEY, name text);
```

```
insert into B (name)
       values ('Rutabaga'),
              ('Pirate'),
              ('Darth Vader'),
              ('Ninja');
select * from A;
select * from B;
```

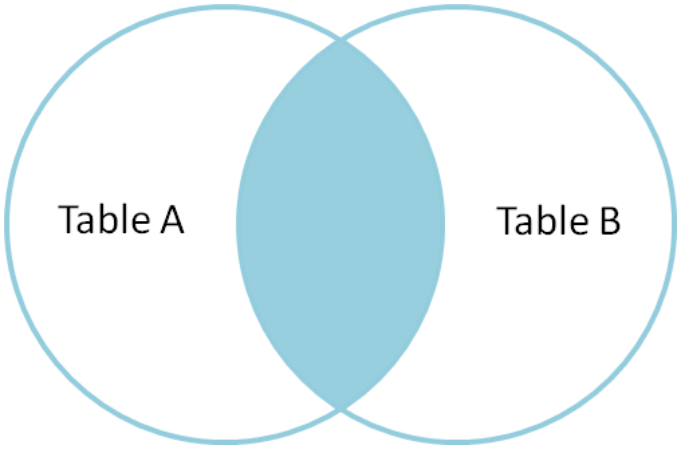
Let's look at several kinds of joins. (There are others, but this will get across the most common types.)

Inner Join

An **inner join** produces the rows for which attributes in **both** tables match. (If you just say `JOIN` in SQL, you get an inner join; the word `INNER` is optional.)

```
select * from A INNER JOIN B on A.name = B.name;
```

We think of the selection done by the `on` condition as a *set operation* on the rows of the two tables. Specifically, an inner join is akin to an intersection:

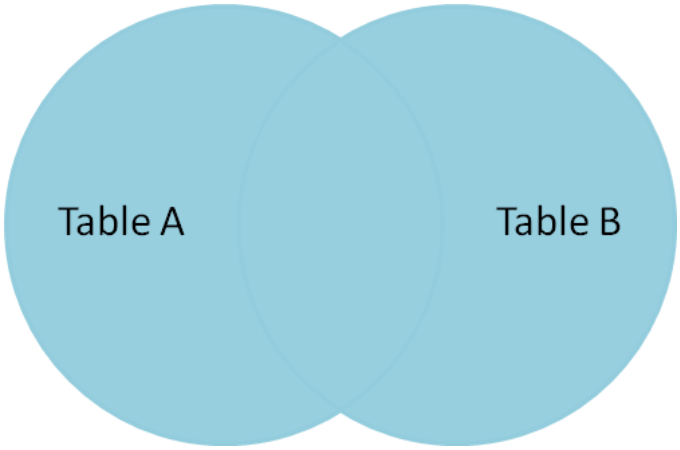


Full Outer Join

A full outer join produces the full set of rows in **all** tables, matching where possible but `null` otherwise.

```
select * from A FULL OUTER JOIN B on A.name = B.name;
```

As a set operation, a full outer join is a *union*

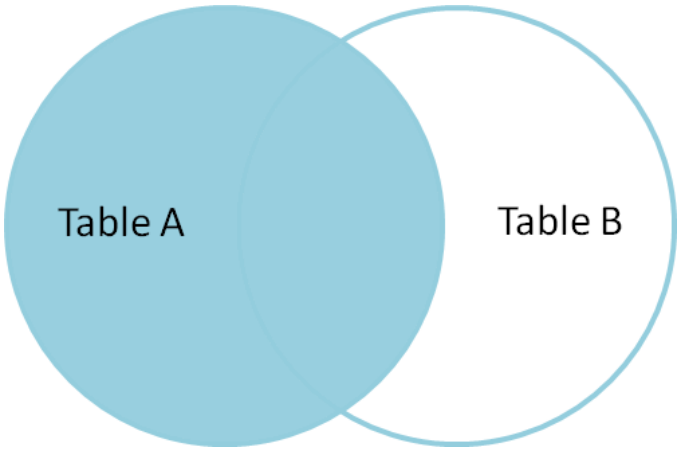


Left Outer Join

A left outer join produces all the rows from A, the table on the "left" side of the `join` operator, along with matching rows from B if available, or `null` otherwise. (`LEFT JOIN` is a shorthand for `LEFT OUTER JOIN` in postgresql.)

```
1 select * from A LEFT OUTER JOIN B on A.name = B.name;
```

A left outer join is a hybrid set operation that looks like:

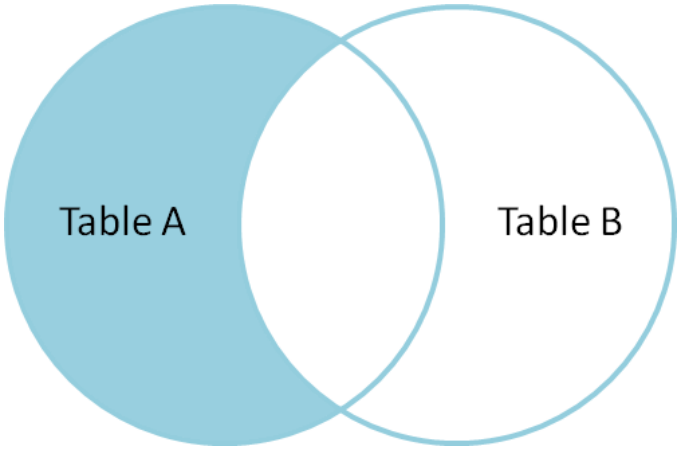


Set Difference

Exercise: Give a selection that gives all the rows of A that are **not** in B.

```
1 select * from A LEFT OUTER JOIN B on A.name = B.name where B.id IS null;
```

This corresponds to a *set difference* operation A - B:

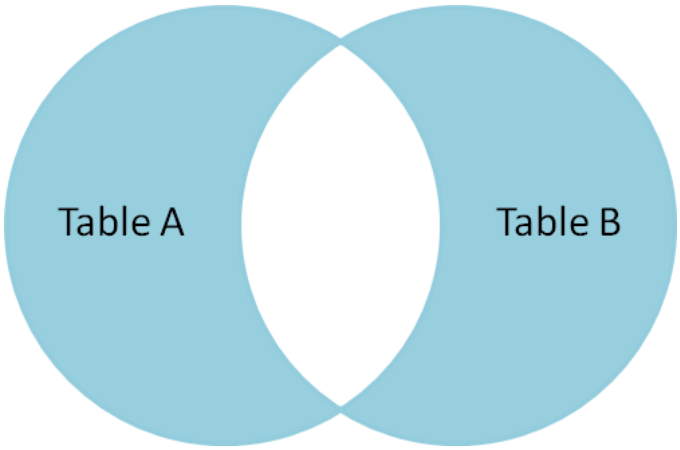


Symmetric Difference

Exercise: Select the rows of A not in B *and* the rows of B not in A.

```
select * from A FULL OUTER JOIN B on A.name = B.name
    where B.id IS null OR A.id IS null;
```

This is the set operation known as a symmetric difference, $A \triangle B = (A - B) \cup (B - A)$:



A slightly more meaningful example

Exercise: Using the `cities` and `countries` tables we created earlier, do the following:

1. List city name, postal code, and country name.

```
select name, postal_code, country_name
    from cities inner join countries
    on cities.country_code = countries.country_code;
```

1. List city name, country, and address as a valid string.

```
select cities.name as city, country_name as country,
       concat(name, ', ', country_name, ' ', postal_code) as address
    from cities inner join countries
    on cities.country_code = countries.country_code;
```

Notice how we can give new names in the produced table (using `AS`) and we can include new columns derived from the old ones.

More:

```
create table venues (
       id SERIAL PRIMARY KEY,
       name varchar(255),
       street_address text,
       type char(7) CHECK (type in ('public', 'private')) DEFAULT 'public',
       postal_code varchar(9),
       country_code char(2),
       FOREIGN KEY (country_code, postal_code)
         REFERENCES cities (country_code, postal_code) MATCH FULL
);
insert into venues (name, postal_code, country_code)
  values ('Crystal Ballroom', '97205', 'us'),
         ('Voodoo Donuts', '97205', 'us'),
         ('CN Tower', 'M4C185', 'ca');
update venues set type = 'private' where name = 'CN Tower';
select * from venues;
```

Now create a `social_events` table with an automatic id field, a title field that is text and fields starts and ends of type `timestamp`, and a foreign key for the venue id. Populate it with a few social events. (Timestamps look like '2012-02-15 17:30:00'.)

```
create table social_events (
       id SERIAL PRIMARY KEY,
       title text,
       starts timestamp DEFAULT timestamp 'now' + interval '1 month',
       ends timestamp DEFAULT timestamp 'now' + interval '1 month' + interval '3 hours',
       venue_id integer REFERENCES venues (id)
);
insert into social_events (title, venue_id) values ('LARP Club', 3);
insert into social_events (title, starts, ends)
  values ('Fight Club', timestamp 'now' + interval '12 hours', timestamp 'now' + interval '16 hours
insert into social_events (title, venue_id)
  values ('Arbor Day Party', 1), ('Doughnut Dash', 2);
select * from social_events;
```

Exercise: List a) all social events with a venue with the venu names, and b) all social events with venue names even if missing.

```
select e.title as event, v.name as venue FROM social_events e JOIN venues v
  on e.venue_id = v.id;
select e.title as event, v.name as venue FROM social_events e LEFT JOIN venues v
  on e.venue_id = v.id;
```

(Recall that JOIN by itself is a shortcut for INNER JOIN, and LEFT JOIN is a shortcut for LEFT OUTER JOIN.)

When we know we will search on certain fields regularly, it can be helpful to create an **index**, which speeds up those particular searches.

```
create index social_events_title  on social_events using hash(title);
create index social_events_starts on social_events using btree(starts);

select * from social_events where title = 'Fight Club';
select * from social_events where starts >= '2015-11-28';
```

Exercise

Using the `ALTER TABLE` command, add a text 'organizer' column to the `social_events` table. Add a State/Province column to `cities` table.

Then update `venues` with street addresses, and use a join to create full address labels for mailing the organizer of each event, e.g.

Tyler Durden Organizer: Fight Club 100 Warehouse Road Portland, Oregon 97205 United States

Exercise

We will use the `personae`, `elements`, and `courses` tables defined in `personae-elements.sql` from the from the documents repository. Alter the `events` table so that the `persona` and `element` columns are foreign keys into these new tables.

```
alter table events ADD FOREIGN KEY (persona) REFERENCES personae;
alter table events ADD FOREIGN KEY (element) REFERENCES elements;
```

Then use a join to display student names, course numbers (in the form '<department>-<catalog$_{\text{number}}$>'), scores, number of hints, and the date (in format like 'Thu 26 Mar 2015' if possible) for events after 26 March 2015 at 8am.

```
select p.lastname, p.firstname,
         c.department || '-' || c.catalog_number as course,
         score,
         hints,
         to_char(moment, 'Dy DD Mon YYYY')
       from events
       join personae as p on persona = p.id
       join courses as c on p.course = c.id
       where moment > timestamp '2015-03-26 08:00:00';
```

Using RDBs from a Programming Language

It's nice to be able to type queries into `psql` and see results, but most often you'd like to do more than that. You're not just making a database to run handwritten queries – you're using it to store data for a big project, and that data then needs to be used to fit models, make plots, prepare reports, and all sorts of other useful things. Or perhaps your code is *generating* data which needs to be stored in a database for later use.

Regardless, you'd like to run queries inside R, Python, or your preferred programming language, and get the results back in a form that can easily be manipulated and used.

Fortunately, PostgreSQL – and most other SQL database systems – use the *client-server* model of database access. The database is a *server*, accessible to any program on the local machine (like the `psql` client) and even to programs on other machines, if the firewall allows it.

This is why you need a password to start `psql`. `psql` connects to the running Postgres server, and to do so it needs a username (the user you logged into SSH as) and a password.

But this also means you can run scripts on your own computer which connect to Postgres with that same username and password.

SQL in R

The RPostgreSQL package provides the interface you need to connect to Postgres from within R. There are similar packages for other database systems, all using a similar interface called DBI, so you can switch to MySQL or MS SQL without changing much code.

To start using Postgres from within R, you need to create a *connection* object, which represents your connection to the server.

```
library(RPostgreSQL)

con <- dbConnect(PostgreSQL(), user="yourusername", password="yourpassword",
                 dbname="yourusername", host="pg.stat.cmu.edu")
```

`con` now represents the connection to Postgres. Queries can be sent over this connection. You can connect to multiple different databases and send them different queries.

To send a query, use `dbSendQuery`:

```
result <- dbSendQuery(con, "SELECT persona, score FROM events WHERE ...")
```

`result` is an object representing the result, but *does not* load the actual results all at once. If the query result is very big, you may want to only look at chunks of it at a time; otherwise, you can load the whole thing into a data frame. `dbFetch` loads the requested number of rows from the result, or defaults to loading the entire result if you'd prefer, all in a data frame.

```
data <- dbFetch(result) # load all data

data <- dbFetch(result, n=10) # load only ten rows

dbClearResult(result)
```

As a shortcut, `dbGetQuery` runs a query, fetches all of its results, and clears the result, all in one step.

SQL in Python

Psycopg is a popular PostgreSQL package for Python. It has a different interface; since Python doesn't have native data frames, you can instead iterate over the result rows, where each row is a tuple of the columns. To connect:

```
import psycopg2

conn = psycopg2.connect(host="pg.stat.cmu.edu", database="yourusername",
                        user="yourusername", password="yourpassword")

cur = conn.cursor()

cur.execute("INSERT INTO foo (bar, baz, spam) "
            "VALUES (17, 'walrus', 'penguin')")
```

If we do a `SELECT`, we can get the results with a `for` loop or the `fetchone` and `fetchmany` methods:

```
cur.execute("SELECT * FROM events")

# iterating:
for row in cur:
    print(row)

# instead, one at a time:
row = cur.fetchone()
```

The `execute` method is used regardless of the type of query.

SQL in Other languages

Most modern programming languages have libraries for interfacing with an SQL server. The behavior and organization is usually very similar to those in R and Python. Some (such as Ruby and the modern lisps) offer more syntactic integration – effectively, language constructs that capture SQL structure – that can be a pleasure to use.

A Brief Interlude on Practicing Safe SQL

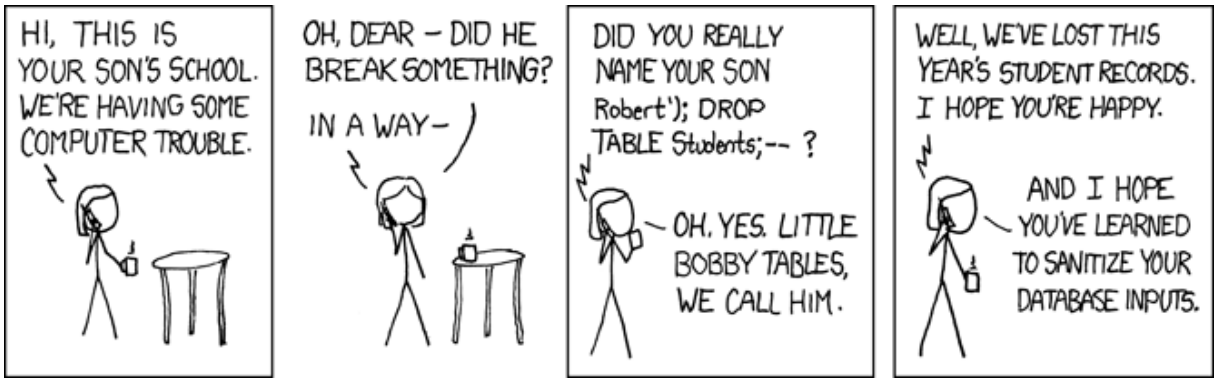
Suppose you've loaded some data from an external source – a CSV file, input from a user, from a website, another database, wherever. You need to use some of this data to do a SQL query.

```
result <- dbSendQuery(paste0("SELECT * FROM users WHERE username = '", username, "' ",
                             "AND password = '", password, "'"))
```

Now suppose `username` is the string "'; DROP TABLE users;–". What does the query look like before we send it to Postgres?

```
SELECT * FROM users
WHERE username = ''; DROP TABLE users; -- AND password = 'theirpassword'
```

We have *injected* a new SQL statement, which drops the table. Because `--` represents a comment in SQL, the commands following are not executed.



Less maliciously, the username might contain a single quote, confusing Postgres about where the string ends and causing syntax errors. Or any number of other weird characters which mess up the query. Clever attackers can use SQL injection to do all kinds of things – imagine if the `password` variable were `foo' OR 1=1` – we'd be able to log in regardless of if the password is correct.

We need a better way of writing queries with parameters determined by the code. Fortunately, database systems provide *parametrized queries*, where the database software is explicitly told "this is an input, with this value" so it knows not to treat it as SQL syntax. For example:

```
username <- "'; DROP TABLE users;--"
password <- "walruses"

query <- sqlInterpolate(con,
                        "SELECT * FROM users WHERE username = ?user AND password = ?pass",
                        user=username, pass=password)

users <- dbGetQuery(con, query)
```

Strings of the form `?var` are replaced with the corresponding `var` in the arguments, but with any special characters escaped so they do not affect the meaning of the query. In this example, `query` is now

```
SELECT * FROM users WHERE username = '''; DROP TABLE users;--'
AND password = 'walruses'
```

Note how the single quote at the beginning of `username` is doubled there: that's a standard way of escaping quotation marks, so Postgres recognizes it's a quote inside a string, not the boundary of the string.

psycopg2 provides similar facilities:

```
cur.execute("SELECT * FROM users "
            "WHERE username = %(user)s AND password = %(pass)s",
            {"user": username, "pass": password})
```

You should *always* use this approach to insert data into SQL queries. You may think it's safe with your data, but at the least opportune moment, you'll encounter nasal demons.

Appendix: A Few Advanced Maneuvers, Part I

Subqueries

You can use select query in `()`'s within a `WHERE` clause:

```
select title, starts from social_events where
     venue_id in (select id from venues where name ~ 'room');
```

There are various other functions/operators that can be used on subqueries as well, such as `in`, `not in`, `exists`, `any`, `all`, and `some`.

For example, this looks like an inner join:

```
select title, starts from social_events where
     exists (select 1 from venues where id = social_events.venue_id);
```

How would you do this with a join?

Aggregate Functions and Grouping

Aggregate functions operate on one or more attributes to produce a summary value. Examples: `count`, `max`, `min`, `sum`.

Add Pittsburgh (and perhaps some other cities) to the cities table. Add two or more Pittsburgh venues and one or more social events at each of those venues.

Let's count the number of social events at one of those venues:

```
select count(*) from social_events where venue_id = 4;
```

Exercise: Given a city name (like 'Pittburgh'), count the total number of social events within that city.

We often want to apply aggregate functions not just to whole columns but to **groups of rows** within columns. This is the province of the `GROUP BY` clause.

We could write

```
select count(*) from events where venue_id = 1;
select count(*) from events where venue_id = 2;
select count(*) from events where venue_id = 3;
select count(*) from events where venue_id = 4;
```

But that is tedious and gives four separate scalars.

Instead, we use the `GROUP BY` modifier:

```
select venue_id, count(*) from events group by venue_id;
```

You can apply conditions on grouped queries. Instead of `WHERE` for those conditions, you use `HAVING`, with otherwise the same syntax. Short version: `WHERE` select rows, and `HAVING` selects groups.

```
select venue_id, count(*) from events group by venue_id
    having venue_id is not NULL;
select venue_id, count(*) from events group by venue_id
    having count(*) > 1 AND venue_id is not NULL;
```

Window Functions

Aggregate functions let you summarize mulitiple rows, but they summarize *with a single value for each group*. **Window functions** are similar except they let us **tag each row with the summary**.

(Make sure you have at least one venue with more than one social event in the example to follow. For instance,

```
insert into social_events (title, venue_id)
  values ('Valentine''s Day Party', 1), ('April Fool''s Day Party', 1);
```

Notice the double ” to escape the single quote.)

Now, compare

```
select venue_id, count(*) from social_events group by venue_id
    order by venue_id;
select venue_id, count(*) OVER (PARTITION BY venue_id)
    from social_events order by venue_id;
```