PostGIS and Rails

(the hard way)

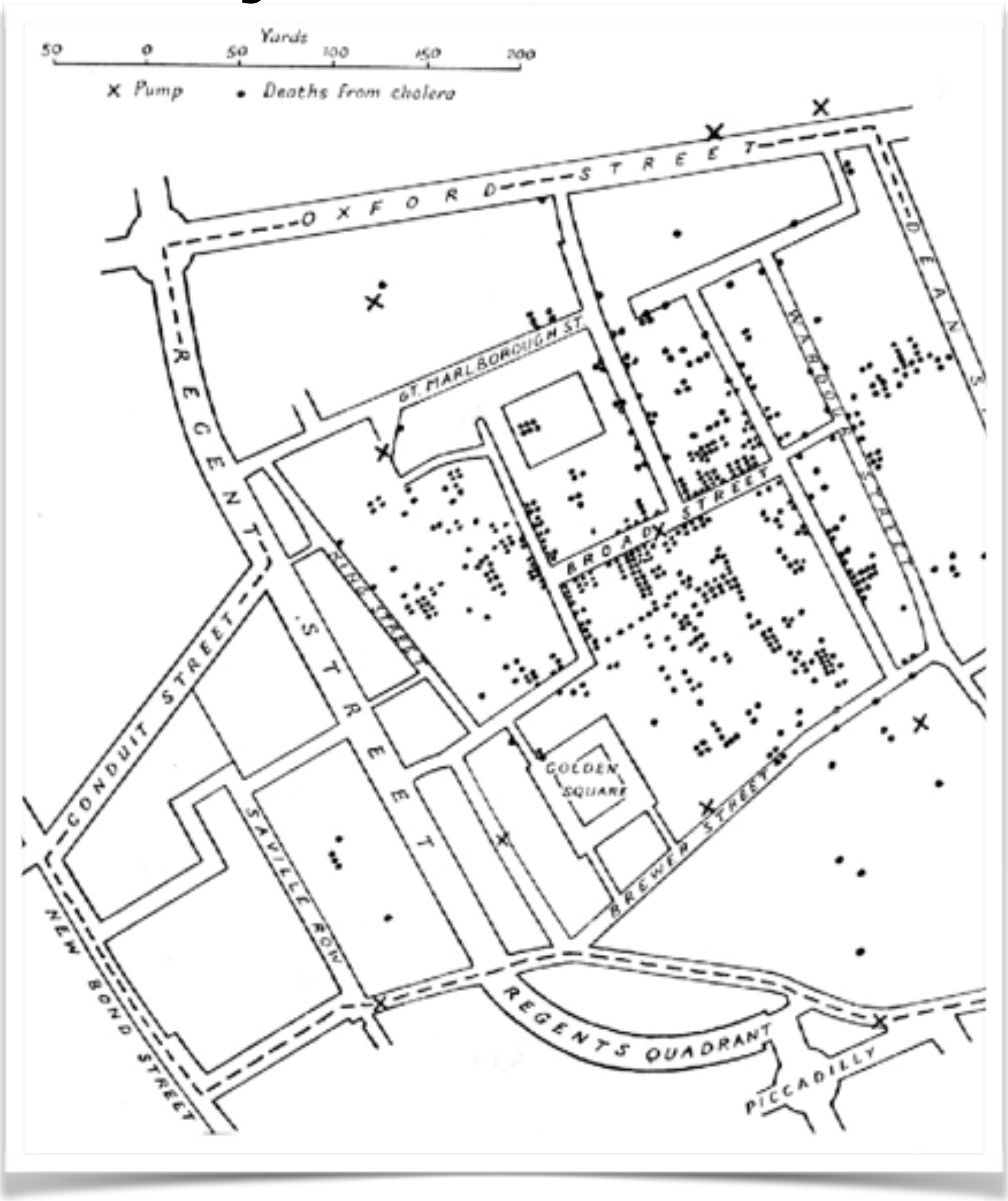
Presented to Cincy.rb
19May2015 – Ryan Dlugosz (@lbwski)

GIS – Geographic Information System

- Store, manipulate and analyze Geographic data

- People go to school for this stuff (so don't be surprised when you feel in over your head)

- Census Bureau, Energy Sector, Government major employers of GIS



You might not need this!

- Google Places API
- Open Street Maps API
- Foursquare, Yelp, etc APIs

If you just need to map something, or find out what is nearby your location, using an API is probably your best bet!

PostGIS

- Spatial extensions for PostgreSQL
  - POINTs, LINEs, POLYGONs
  - Distance, area, length, union, intersection, etc.
- Lets you do things like:

```
SELECT superhero.name
FROM   city, superhero
WHERE  ST_Contains(city.geom, superhero.geom)
AND    city.name = 'Gotham';
```

Installing PostGIS

- Homebrew:

  -
    ```
    brew install postgis
    ```
  - …and then:
    ```
    brew reinstall postgresql postgis
    ```
    (sometimes your PostGIS libraries will break when you upgrade things; this often fixes the problem)
- Postgres.app is also a good option & includes PostGIS out of the box!

Adding to Rails!

- **RGeo**
  - Geospatial data for Ruby. Most “PostGIS and Rails” tutorials will have you install this.
  - Allows you to manipulate geometric data in Ruby, just like PostGIS does for SQL.
  - Provides multiple database adapters for ActiveRecord (migrations, mapping to real Ruby types, etc.)
  - *I do not recommend starting here!* Unless you plan on doing a lot of spatial analysis in Ruby (vs in-database) it can be overkill. Better to learn PostGIS and let the database do all the heavy lifting and let your app make use of the *results* of the spatial analysis.
  - https://github.com/rgeo/rgeo

NOT Adding to Rails!

- Use PostGIS directly via SQL from ActiveRecord, no special gems required.

- RGeo is great, but ***analogous to using Active Record without knowing SQL if you don't already know PostGIS.***

- Plus, ActiveRecord extensions typically trail Rails development (e.g., the AR PostGIS adapter presently has problems with Rails 4.2). AR is a volatile area of the Rails codebase, so your future upgrade path may include a lot of bumps.

Aside – Geometry vs. Geography

- PostGIS does more than just maps. Can be used to store any 2–, 3– or 4D objects.

- Most functions operate on **GEOMETRY**

- But occasionally you really do want Geography (e.g., distance calculations far apart on globe)

- Generally speaking, the difference is that Geometry calculations are performed on a plane while Geography calculations are performed on a spheroid.

Aside – SRIDs: Spatial Reference IDentifiers

- Map projections
- There are thousands: states each use their own (each optimized for their own location), etc.
- One of the biggest gotchas is making sure you are using the right SRID
- If you get into complex GIS work where accuracy is vital, you should consult with a GIS expert to ensure you aren't doing something bad wrt map projections!

```
<geom> ST_Transform(geom, srid)
```

PostGIS: Engage

```ruby
class AddPostgisExtension < ActiveRecord::Migration
  def change
    enable_extension :postgis
  end
end
```

Rails 4 added the ability to activate a database extension right from a normal migration!

Dump SQL

Since Active Record doesn’t understand PostGIS columns, we need to use SQL dumping instead of the AR-migration based schema.rb

./config/application.rb:

```ruby
module PostgisDemo
  class Application < Rails::Application
    # dump schema as SQL to structure.sql
    config.active_record.schema_format = :sql
    #...
  end
end
```

The RGeo `activerecord-postgis-adapter` lets you avoid this, but at a price.

Nearby Tweets

- Let’s build an app that stores Tweets that are geocoded. The Twitter stream will give us a lat/lng that we will store and we’ll also use that to create a PostGIS geometry column representing that point.

- Then, we’ll add a method to our model that allows us to find *n* Tweets that are nearby a given lat/lng.

Generate Model

```
$ rails g model tweet time tweet_id language
country_code lat:float lng:float
```

```
class CreateTweets < ActiveRecord::Migration
  def change
    create_table :tweets do |t|
      t.string :time
      t.string :tweet_id
      t.string :language
      t.string :country_code
      t.float :lat
      t.float :lng

      t.timestamps null: false
    end
  end
end
```

Nothing special here!

Add Geometry Column

```
$ rails g migration addGeomToTweets
```

```
class AddGeomToTweets < ActiveRecord::Migration
  def change
    reversible do |direction|
      direction.up do
        execute <<-SQL
          ALTER TABLE tweets
          ADD COLUMN geom GEOMETRY(Point, 3857);
        SQL
      end
      direction.down do
        remove_column :tweets, :geom
      end
    end
  end
end
```

Since we aren't using the ActiveRecord extension library, AR doesn't know how to create a PostGIS geometry column. We'll do it ourselves via SQL. Note the handy `#reversible` pattern for this situation.

Add an Index to the Geometry

```ruby
class AddIndexToTweets < ActiveRecord::Migration
  def change
    reversible do |direction|
      direction.up do
        execute <<-SQL
          CREATE INDEX idx_tweets_on_geom
          ON tweets
          USING GIST (geom);
        SQL
      end
      direction.down do
        remove_index :tweets, name: :idx_tweets_on_geom
      end
    end
  end
end
```

GIST indexes can handle spatial data and many other non-standard data types. Useful for lots of things! Being able to index our spatial data is key to performant queries.

…add a Trigger (sorry) to keep the Geom column in sync with the lat/lng columns

```
CREATE FUNCTION update_tweet_geom() RETURNS trigger
    LANGUAGE plpgsql
    AS $$
        BEGIN
          NEW.geom := ST_GeomFromEWKT('SRID=3857;POINT (' || NEW.lng || ' '
|| NEW.lat || ')');
          RETURN NEW;
        END;
      $$;

CREATE TRIGGER trigger_update_geom_on_tweet_update
BEFORE INSERT OR UPDATE ON tweets
FOR EACH ROW EXECUTE PROCEDURE update_tweet_geom();
```

Avoiding stuff like this is where RGeo shows its benefits…
but you only need to do it once!

As Scott pointed out, you could do this via an `#after_save` callback, but the trigger approach covers the case of mass data imports separate from your app, which is a common thing to do when working with geo data.

Finally… SQL to find the nearby tweets!

- Let’s assume we’ve captured and loaded a bunch of tweets into our database.

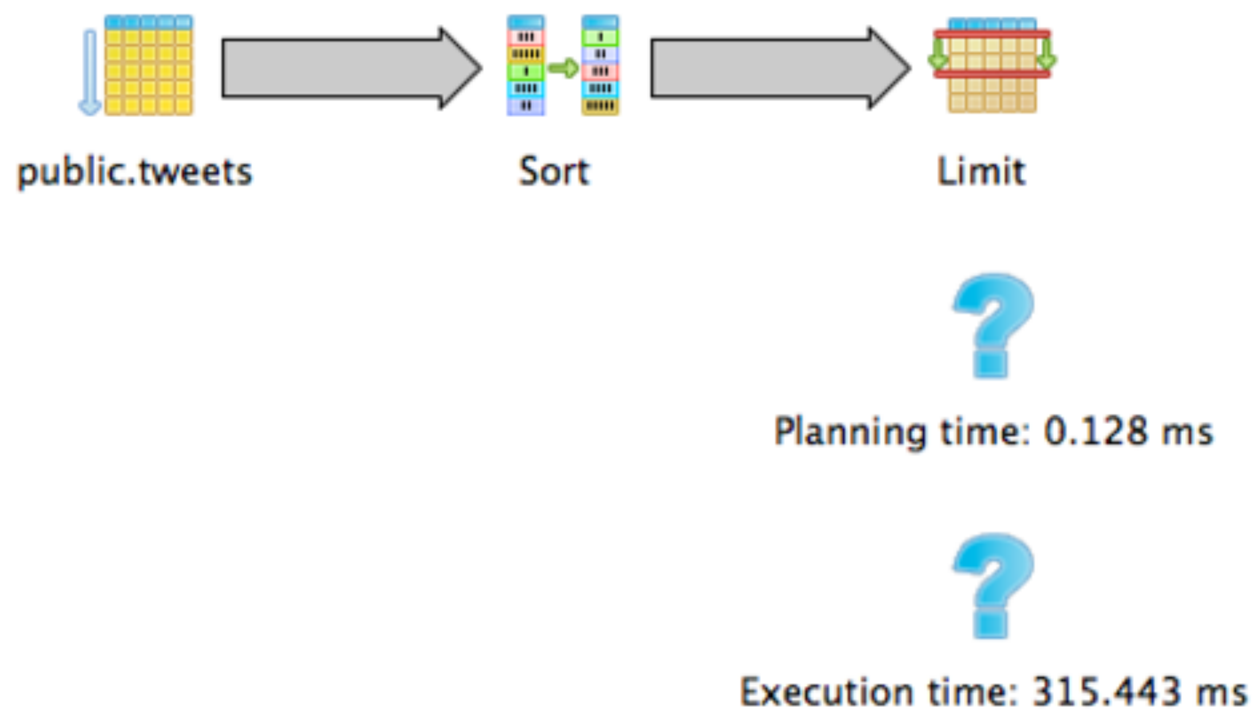
- A query to find the 5 tweets closest to our position can be as simple as:

```
SELECT *,
 ST_Distance(geom,
  ST_GeomFromEWKT('SRID=3857;POINT (-84.3761 39.2472)’))
  as distance
 FROM tweets
 ORDER BY distance ASC
 LIMIT 5;
```

Works, but it’s slow! Let’s use EXPLAIN to understand why…

Find Nearby Tweets (slow)

```
 SELECT *,
  ST_Distance(geom, ST_GeomFromEWKT('SRID=3857;POINT
(-84.3761 39.2472)')) as distance
  FROM tweets
  ORDER BY distance ASC
  LIMIT 5;
```



Planning time: 0.128 ms

Execution time: 315.443 ms

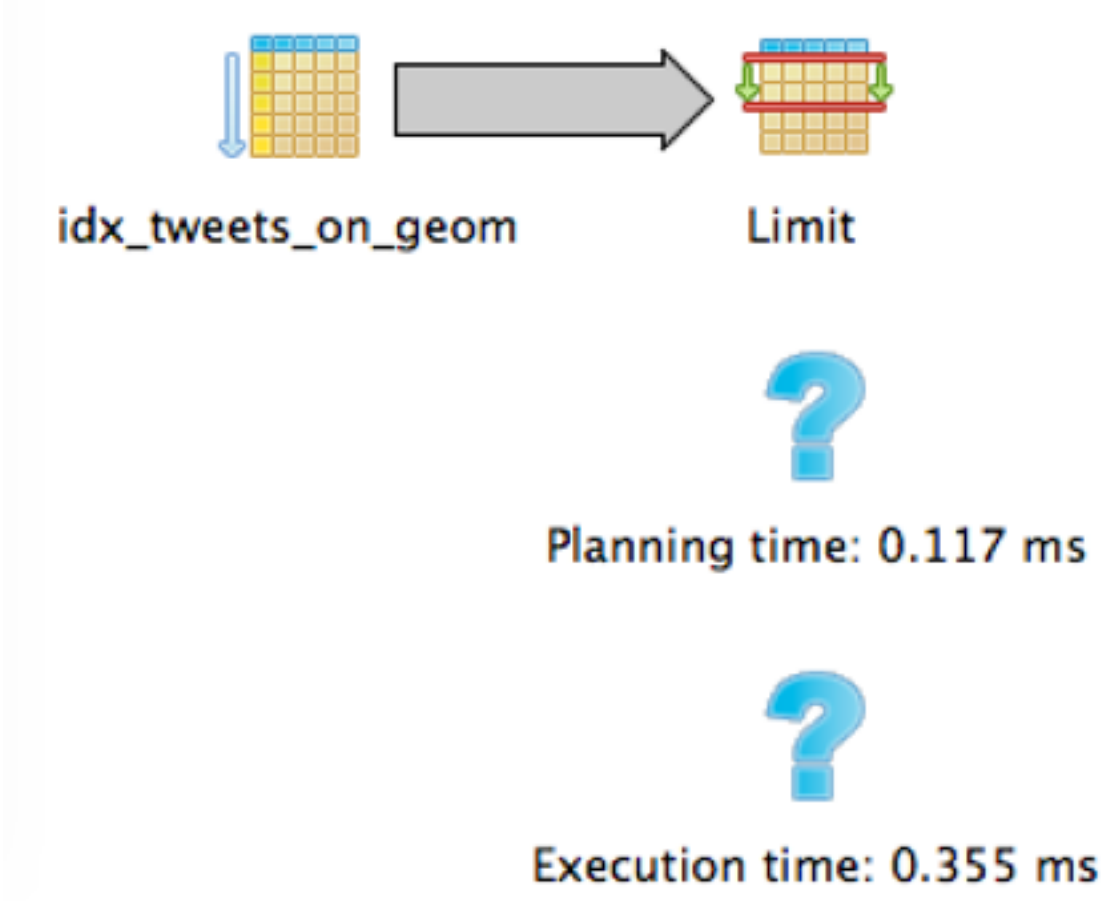
Execution Time: 315ms

Slow!! Has to calculate the distance of EVERY tweet!

Some functions (like ST_Distance) cannot use an index. Therefore, you must either reduce your search space (consider first finding tweets inside a rectangle surrounding your target via ST_Within (which *can* use an index) and then using ST_Distance), or… you can use the KNN operator (see next slide!).

Find Nearby Tweets - KNN

```
 SELECT *,
  geom <-> ST_GeomFromEWKT('SRID=3857;POINT (-84.3761
39.2472)') as distance
  FROM tweets
  ORDER BY distance
  LIMIT 10;
```



Planning time: 0.117 ms

Execution time: 0.355 ms

Execution Time: 0.35ms
(nearly 100x faster!)

FAST!! Can make use of spatial index!

Even better for “finding things near a point” is the KNN (*k* Nearest Neighbors) operator **<->**. It can make use of your spatial index and is rather fast. Introduced in PostGIS 2.1.

Add SQL to Model

- Now that we have our query, we can add it to our Rails Model for easy use within our application.

- Bare SQL in your app can be both *dangerous* (because of sql injection risks if done improperly) and difficult for junior developers to understand. Be sure you are doing this the Right Way. (see: http://rails-sqli.org/)

- The flip side is that in most apps, these geospatial queries are a small portion of the overall code in the app and can be isolated into a separate class that is rarely seen or changed.

Add SQL to Model

```
class Tweet < ActiveRecord::Base
  def self.nearby(lat: 39.2472, lng: -84.3761, count: 5)
    sql = <<-SQL
      SELECT *,
        geom <-> ST_GeomFromEWKT('SRID=3857;POINT (:lng :lat)') as
distance
      FROM tweets
      ORDER BY distance
      LIMIT :count;
    SQL

    Tweet.find_by_sql([sql, { lat: lat, lng: lng, count: count }])
  end
end
```

A simple example, but this is how you could include a raw SQL query in your AR model. You may wish to exclude the Geom column in a production app since it can be large and is of little use to your Ruby code (you're interested in the results calculated via the Geom column; not the column itself).

Note the #find_by_sql syntax used here; this is the safe way to include potentially dangerous strings in your SQL. Again, in a real app you would want to consider abstracting this away into a separate class for clarity/ safety from devs not familiar with PostGIS.

Now, we can use it in our app!

```
irb(main):022:0> Tweet.count
   (52.0ms)  SELECT COUNT(*) FROM "tweets"
=> 340528

irb(main):023:0> Tweet.nearby(lat: 39, lng: -84, count: 5).map(&:distance)
  Tweet Load (2.0ms)        SELECT *,
        geom <-> ST_GeomFromEWKT('SRID=3857;POINT (-84 39)') as distance
      FROM tweets
      ORDER BY distance
      LIMIT 5;

=> [0.0698045989795416, 0.129378466238647, 0.290246518133419, 0.299941772955311,
0.30044666365637]
```

Woot!

Handy Functions

- ST_Translate - Convert SRIDs
- ST_GeomFromEWKT - Geometry from text
- ST_Point - Create a Point
- ST_AsText - Describe geometry as text
- ST_GeoHash - GeoHash representation of Geometry
- ST_AsGeoJSON - GeoJSON of geometry (drop onto a map)
- <-> KNN distance operator
- ST_Distance, ST_Length, ST_Overlaps, ST_Within…

Performance Tips

- Make *sure* your queries are using an index! Geo data is large, which makes table scans extra costly. Also, geo data often involves row counts into the millions.
  (EXPLAIN is your friend - use it on every query.)

- Always look to make your search space as **small** as possible and as **simple** as possible (e.g., bounding boxes and lower-res polygons)

- CLUSTER your data if possible (re-writes the data on disk in order of an arbitrary query, e.g., use CLUSTER with a GeoHash of your Geom column to ensure points nearby each other spatially are nearby each other on disk). Especially important if your database is not on an SSD hard drive.

Where to find data?

- http://www.census.gov/geo/maps-data/data/tiger.html <— TONS of data (ZCTAs, boundaries, roads, water, demographics, etc)
    - Tiger data will allow you to set up your own geocoder, etc.
    - Speaking of **geocoding** (converting an address into lat/lng), you can make use of APIs for this should you need it. Google, Open Street Map, etc. There's also a geocoder gem that will integrate the APIs for you.
- http://www.data.gov/, http://www.usgs.gov/
- https://en.wikipedia.org/wiki/List_of_GIS_data_sources
- Each state has it's own repo, as do large cities, e.g., Cincinnati has CAGIS: http://cagismaps.hamilton-co.org/cagisportal
- Scrape your own (e.g., tweet streams, geolocating users, etc.)
- Remember to check / translate SRIDs!

Learning Resources

- Boundless Workshop: http://workshops.boundlessgeo.com/postgis-intro/
- PostGIS Docs: http://postgis.net/documentation
- BostonGIS: http://www.bostongis.com/
- /r/gis
- https://gis.stackexchange.com/
- http://qgis.org/ – QGIS is the OSS that lets you visualize data on a map. Complicated interface and slow but powerful.