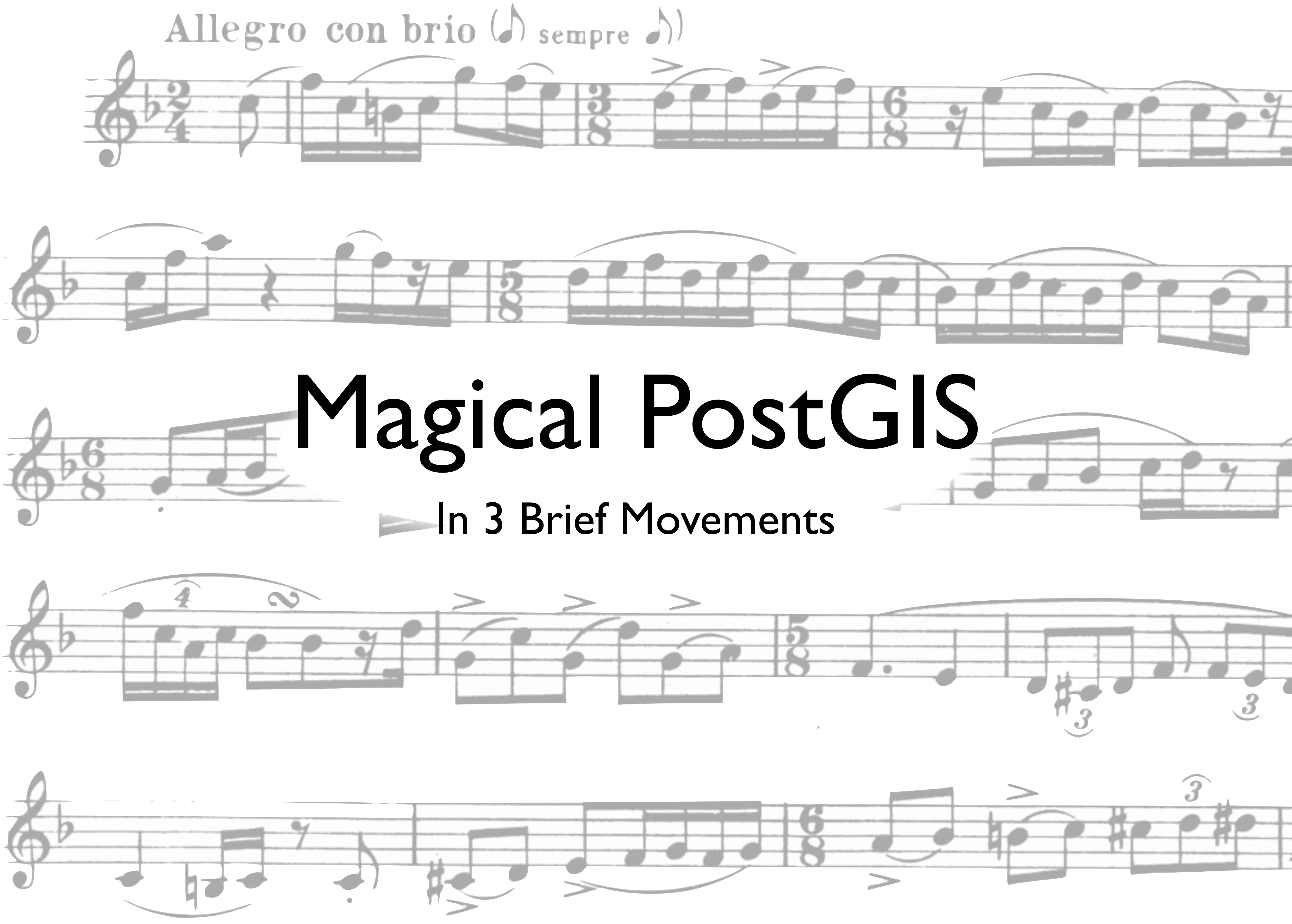


Magical PostGIS

In 3 Brief Movements

CARTODB

Geospatial on the cloud

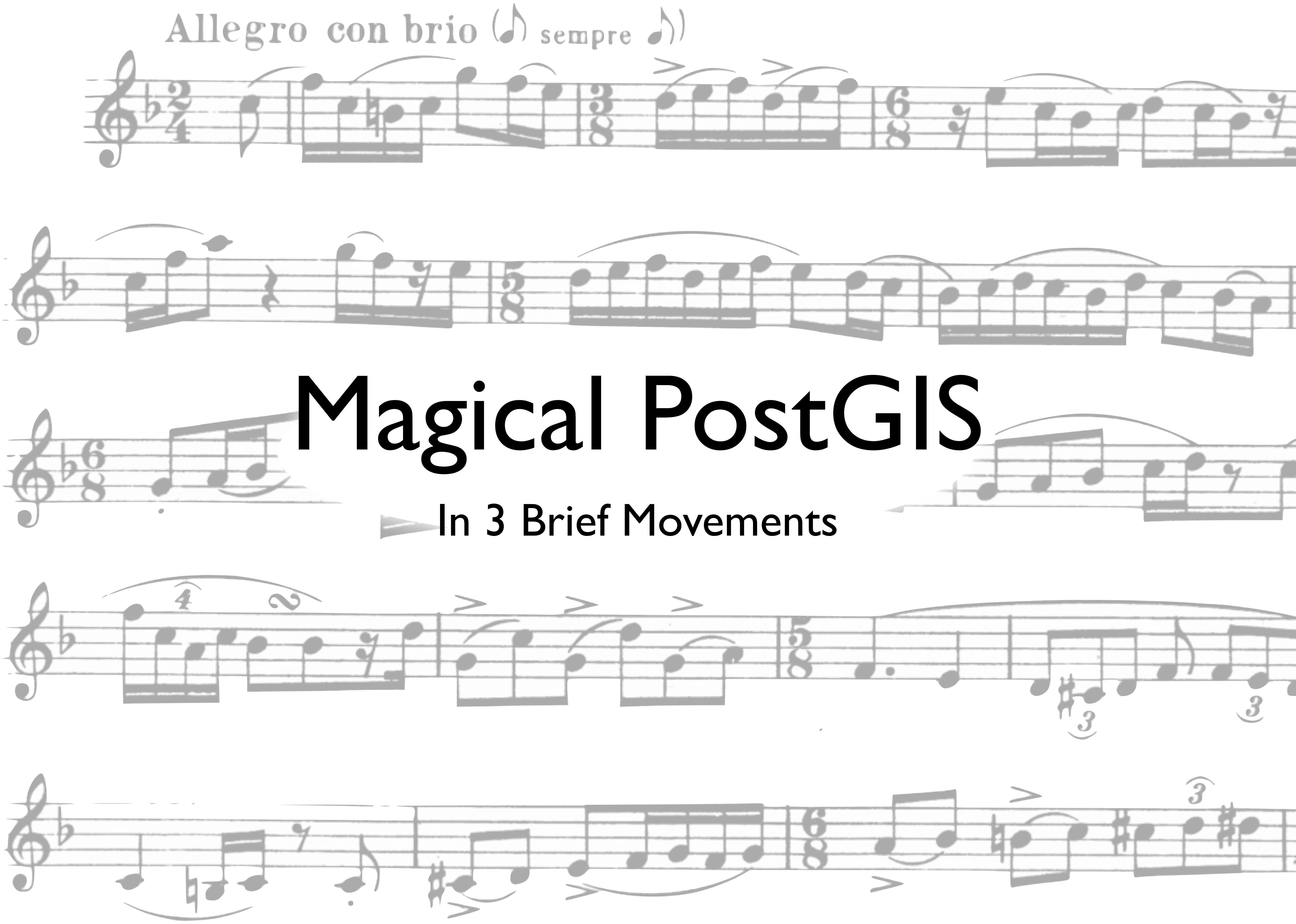
thanks to the good folks at cartodb for employing me
and letting me work on PostGIS.
like a lot of SaaS companies, cartodb has a strong open source ethic
(stronger than most, actually) because their system
is built on top of open source components.
the “DB” in cartodb is actually postgis,
so much of what I’m talking about today can be run in the cartodb cloud,
and some of my examples actually do that.

CARTODB

Geospatial on the cloud

Powered by PostGIS

thanks to the good folks at cartodb for employing me
and letting me work on PostGIS.
like a lot of SaaS companies, cartodb has a strong open source ethic
(stronger than most, actually) because their system
is built on top of open source components.
the “DB” in cartodb is actually postgis,
so much of what I’m talking about today can be run in the cartodb cloud,
and some of my examples actually do that.



Magical PostGIS

In 3 Brief Movements

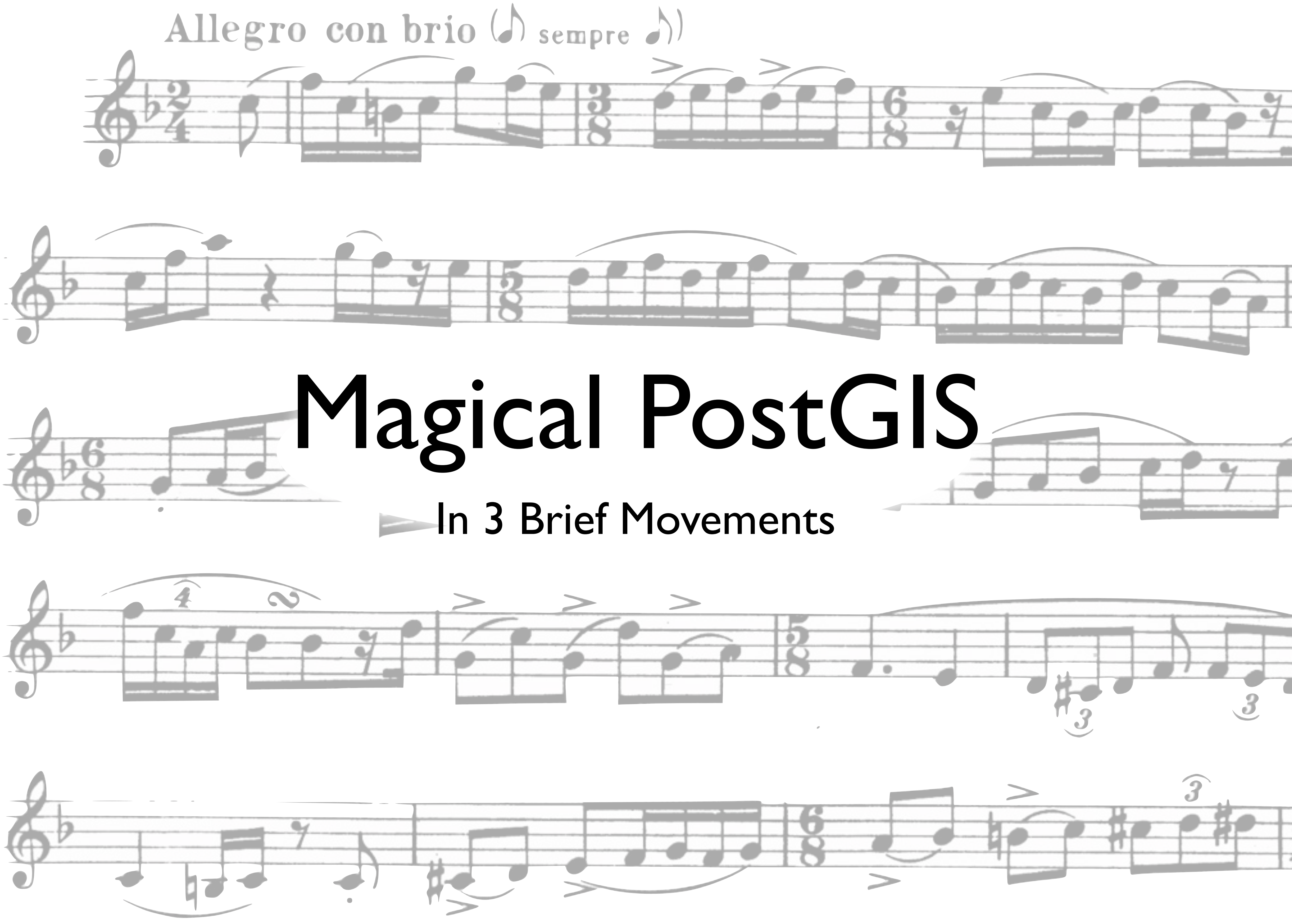
So, this is supposed to be “magical postgis”,
and perhaps the “applesque” name is what drew you to the room,
but in retrospect perhaps I should have called it

For show 'n tell, I brought in my favourite PostgreSQL extensions!
colors
Counting

show and tell, since I seem to have a lot of material about my favourite toys or maybe

Stupid Extension Tricks

stupid extension tricks would have been more honest,
since I’ve got some crazy examples of crazy extensions,
but regardless, on the playbill it is magical postgis in three “brief” movements



Magical PostGIS

In 3 Brief Movements

I wanted to give this talk because I feel like folks aren't appreciating
the kind of deep and beautiful magic that they can create
using little more than their standard backend database.
Too often, people have a utilitarian view of their database.
They really don't like their database at all.

your database is not just a bit bucket

To them, it’s just a bit bucket.
It holds a bunch of tables.
They stuff data in, they drag data out.
Some people hate their database **so much** that they hide it away
behind an Object Relational Mapping layer, an ORM

your ORM doesn't change the nature of the situation

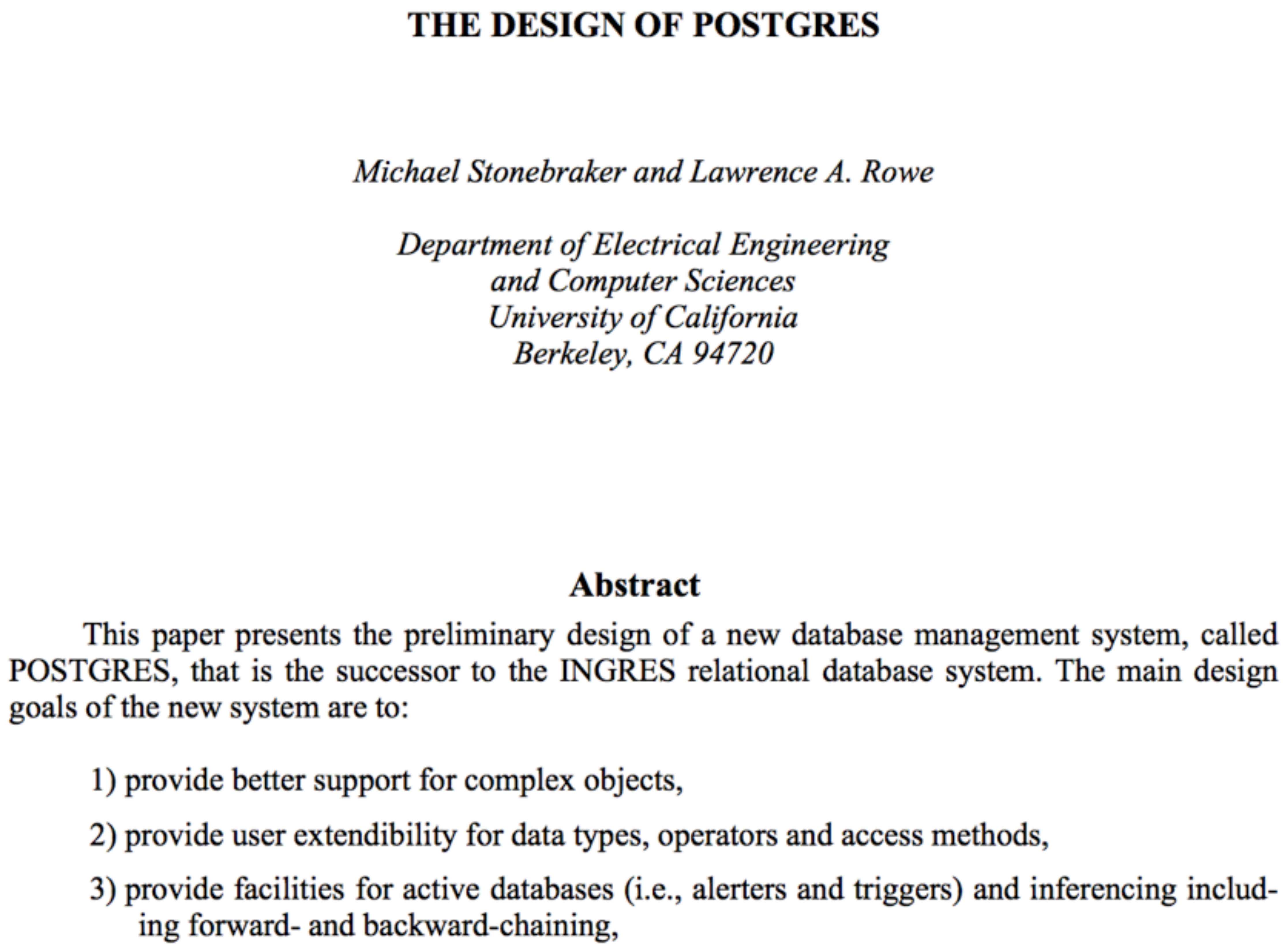
so they can pretend the bit bucket isn't there in the background,
doing the hard work,
so they can pretend they are all alone in their beautiful little middleware language.
And they are really missing out,
because once you get to know it more intimately, you come to realize

your database is beautiful

that your database is a beautiful, beautiful thing.
It’s **not just a bit bucket**,
it’s a magical toolbox with all kinds of good stuff inside.
So this talk is actually not so much about PostGIS,
as about the kinds of things you can do with PostGIS,
when you combine it with the magic that is inside the PostgreSQL database.

Michael
Stonebraker

PostgreSQL is so magical because it was designed from the start
to be more than a bit-bucket.
Michael Stonebraker had already spent a decade pushing around bits
in the Ingres project
when he dreamed up his next generation database in 1986,



THE DESIGN OF POSTGRES

Michael Stonebraker and Lawrence A. Rowe

*Department of Electrical Engineering
and Computer Sciences
University of California
Berkeley, CA 94720*

Abstract

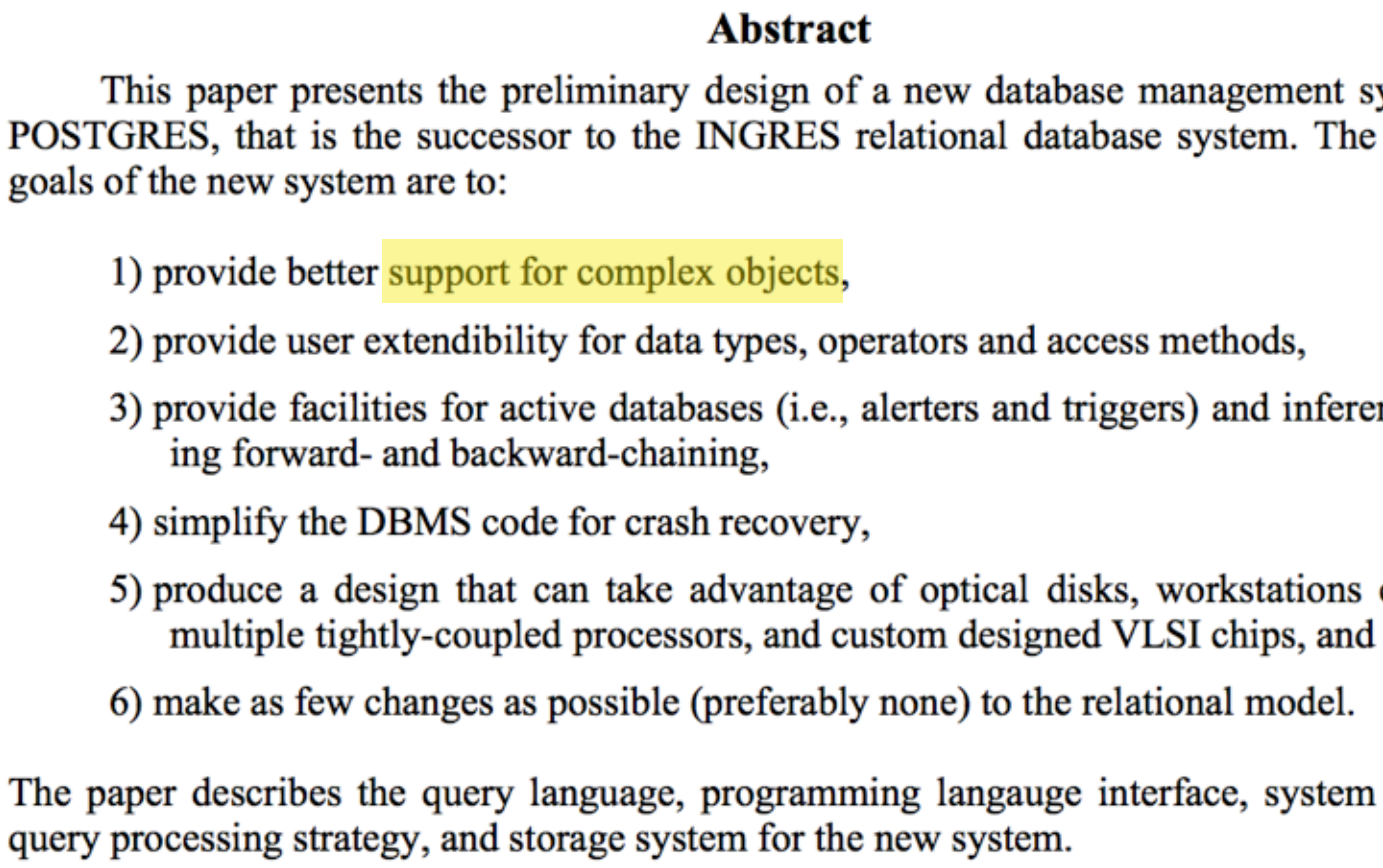
This paper presents the preliminary design of a new database management system, called POSTGRES, that is the successor to the INGRES relational database system. The main design goals of the new system are to:

1) provide better support for complex objects,

2) provide user extendibility for data types, operators and access methods,

3) provide facilities for active databases (i.e., alerters and triggers) and inferencing including forward- and backward-chaining,

and wrote a paper, “the design of postgres”,
which laid out his goals for the new, and at that point, unwritten database.
it’s those goals which form the foundation for PostgreSQL’s awesomeness,
and for PostGIS itself,
in particular



Abstract

This paper presents the preliminary design of a new database management sy POSTGRES, that is the successor to the INGRES relational database system. The goals of the new system are to:

1) provide better support for complex objects,

2) provide user extendibility for data types, operators and access methods,

3) provide facilities for active databases (i.e., alerters and triggers) and inferer ing forward- and backward-chaining,

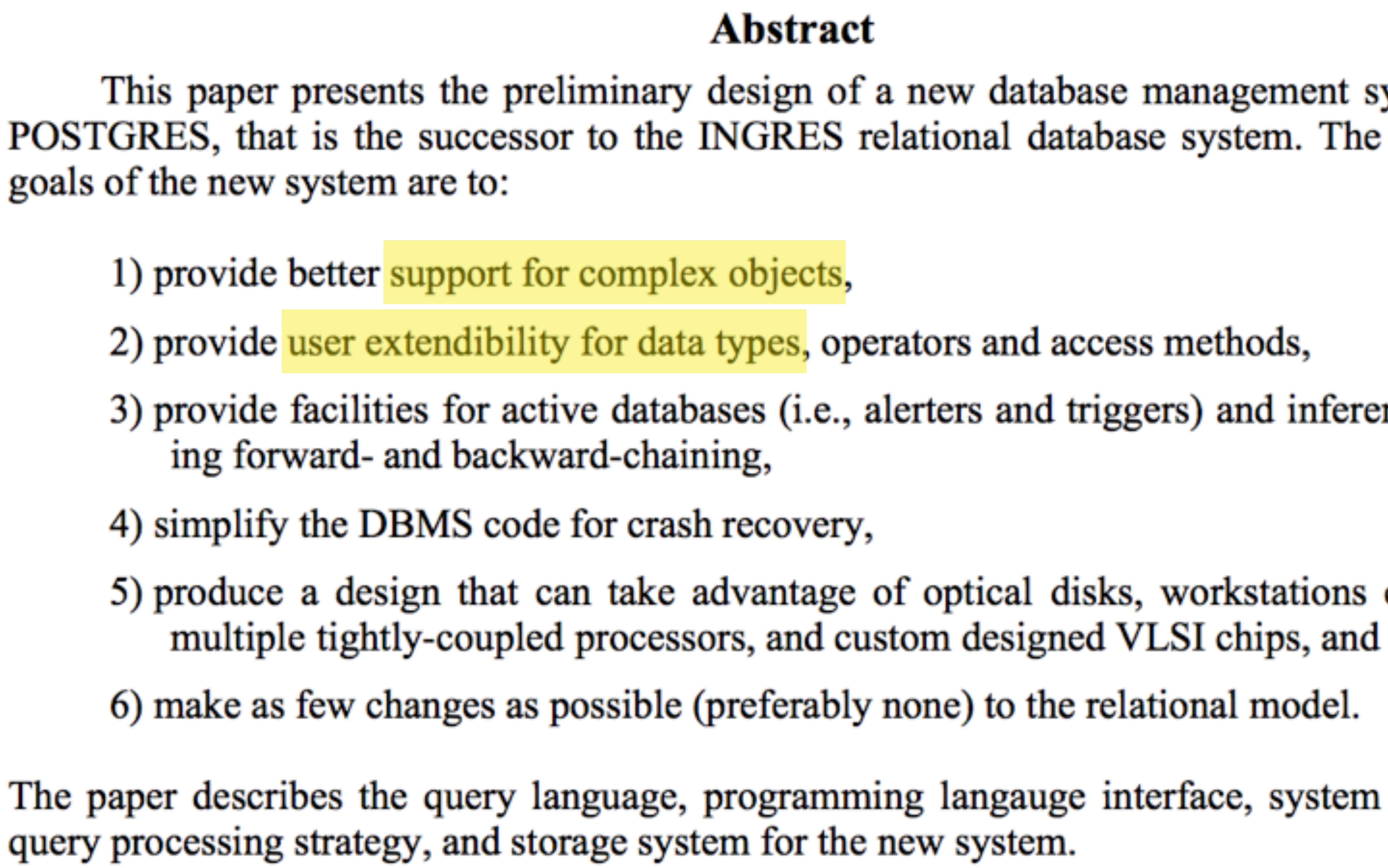
4) simplify the DBMS code for crash recovery,

5) produce a design that can take advantage of optical disks, workstations c multiple tightly-coupled processors, and custom designed VLSI chips, and

6) make as few changes as possible (preferably none) to the relational model.

The paper describes the query language, programming langauge interface, system query processing strategy, and storage system for the new system.

support for complex objects.
geometry and geography are complex objects! so are rasters!
<x> user extensibility is what allows PostGIS to exist at all, it allows anyone to add types and functions to the database at runtime, most of the fun stuff in PostgreSQL takes advantage of extension points
<x> active features are a pretty common database feature now, and they let the database take a hand in managing data flow
<x> and the relational model is what ties everything together,
what makes the system as powerful as it is, every piece of information is a tuple and tuples are collected in tables



Abstract

This paper presents the preliminary design of a new database management sy
POSTGRES, that is the successor to the INGRES relational database system. The
goals of the new system are to:

1) provide better support for complex objects,

2) provide user extendibility for data types, operators and access methods,

3) provide facilities for active databases (i.e., alerters and triggers) and inferer
ing forward- and backward-chaining,

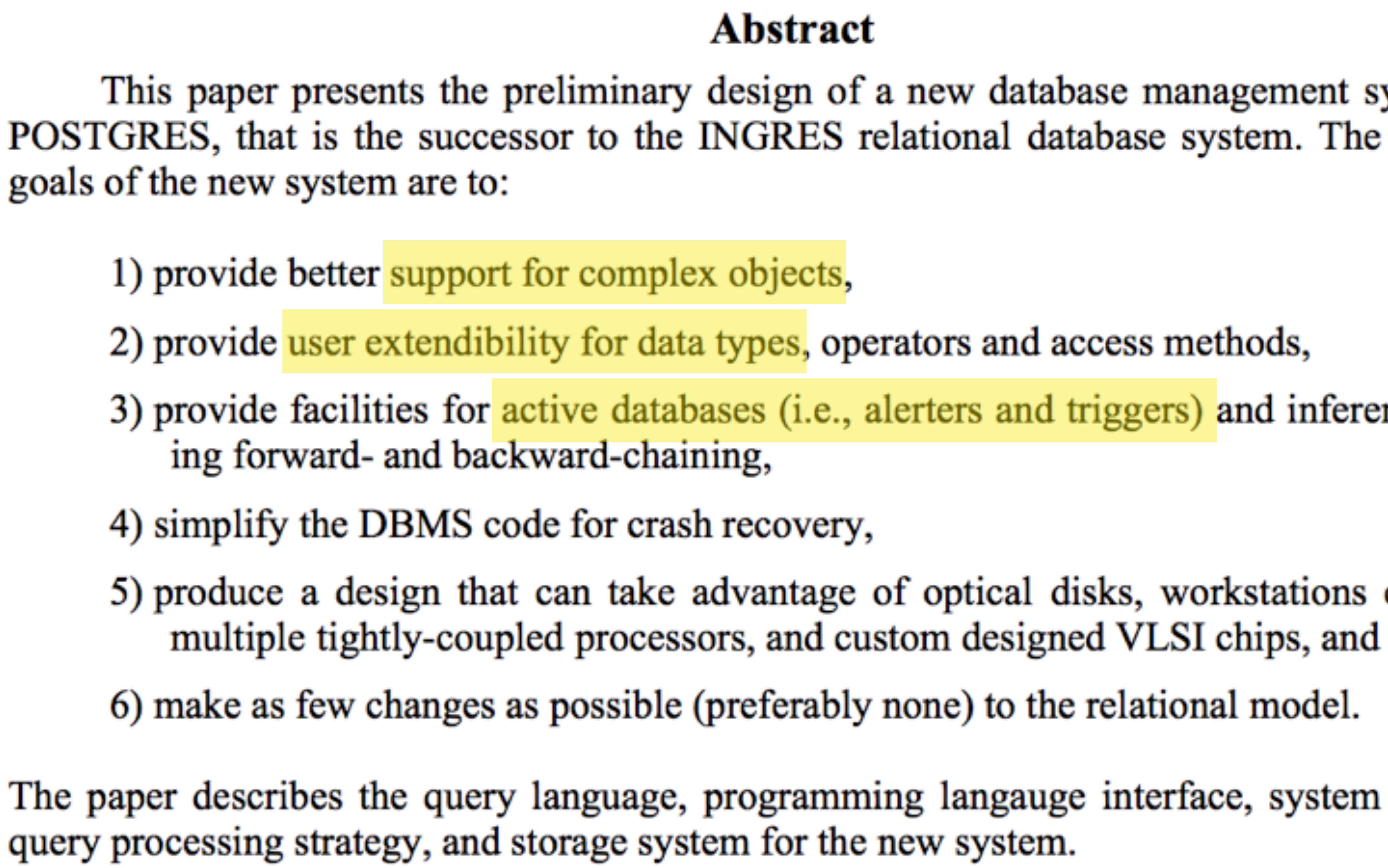
4) simplify the DBMS code for crash recovery,

5) produce a design that can take advantage of optical disks, workstations c
multiple tightly-coupled processors, and custom designed VLSI chips, and

6) make as few changes as possible (preferably none) to the relational model.

The paper describes the query language, programming langauge interface, system
query processing strategy, and storage system for the new system.

support for complex objects.
geometry and geography are complex objects! so are rasters!
<x> user extensibility is what allows PostGIS to exist at all, it allows anyone to add types and functions to the database at runtime, most of the fun stuff in PostgreSQL takes advantage of extension points
<x> active features are a pretty common database feature now, and they let the database take a hand in managing data flow
<x> and the relational model is what ties everything together,
what makes the system as powerful as it is, every piece of information is a tuple and tuples are collected in tables



Abstract

This paper presents the preliminary design of a new database management sy POSTGRES, that is the successor to the INGRES relational database system. The goals of the new system are to:

1) provide better support for complex objects,

2) provide user extendibility for data types, operators and access methods,

3) provide facilities for active databases (i.e., alerters and triggers) and inferen ing forward- and backward-chaining,

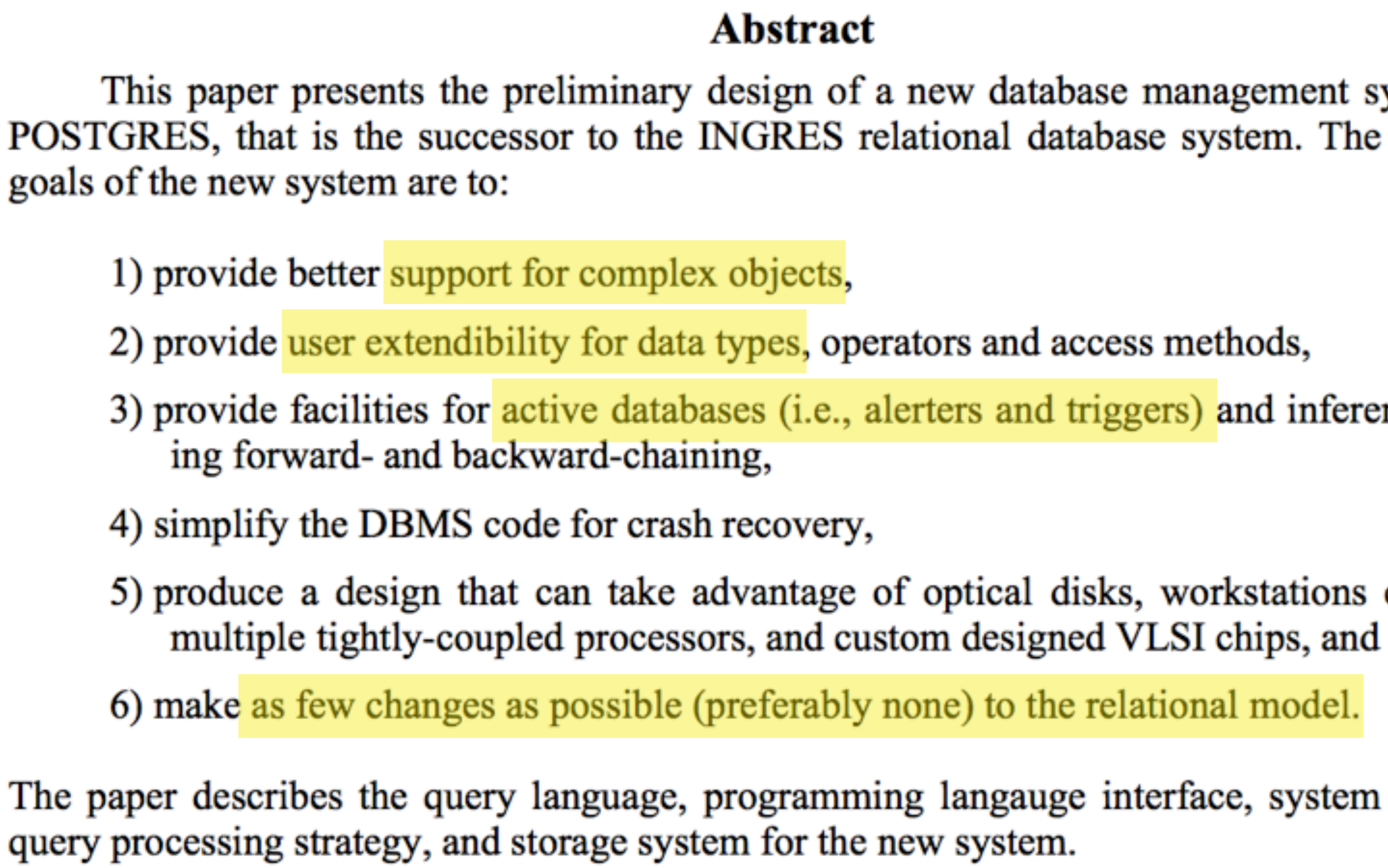
4) simplify the DBMS code for crash recovery,

5) produce a design that can take advantage of optical disks, workstations o multiple tightly-coupled processors, and custom designed VLSI chips, and

6) make as few changes as possible (preferably none) to the relational model.

The paper describes the query language, programming langauge interface, system query processing strategy, and storage system for the new system.

support for complex objects.
geometry and geography are complex objects! so are rasters!
<x> user extensibility is what allows PostGIS to exist at all, it allows anyone to add types and functions to the database at runtime, most of the fun stuff in PostgreSQL takes advantage of extension points
<x> active features are a pretty common database feature now, and they let the database take a hand in managing data flow
<x> and the relational model is what ties everything together,
what makes the system as powerful as it is, every piece of information is a tuple and tuples are collected in tables



Abstract

This paper presents the preliminary design of a new database management sy
POSTGRES, that is the successor to the INGRES relational database system. The
goals of the new system are to:

1) provide better support for complex objects,

2) provide user extendibility for data types, operators and access methods,

3) provide facilities for active databases (i.e., alerters and triggers) and inferen
ing forward- and backward-chaining,

4) simplify the DBMS code for crash recovery,

5) produce a design that can take advantage of optical disks, workstations c
multiple tightly-coupled processors, and custom designed VLSI chips, and

6) make as few changes as possible (preferably none) to the relational model.

The paper describes the query language, programming langauge interface, system
query processing strategy, and storage system for the new system.

support for complex objects.
geometry and geography are complex objects! so are rasters!
<x> user extensibility is what allows PostGIS to exist at all, it allows anyone to add types and functions to the database at runtime, most of the fun stuff in PostgreSQL takes advantage of extension points
<x> active features are a pretty common database feature now, and they let the database take a hand in managing data flow
<x> and the relational model is what ties everything together,
what makes the system as powerful as it is, every piece of information is a tuple and tuples are collected in tables

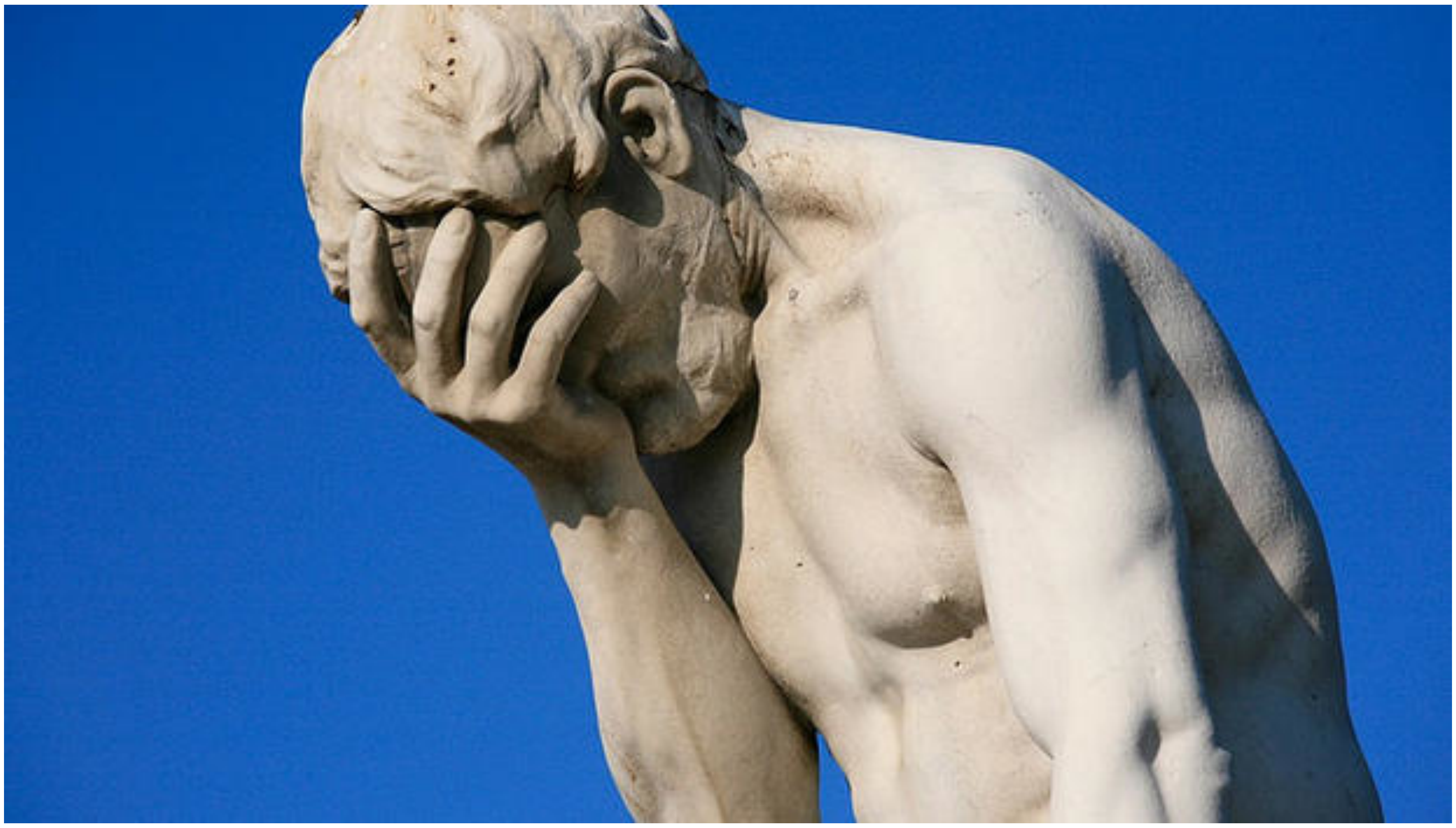
Postgres lived as Stonebraker’s academic project for almost a decade,
but it was useful enough that, by the time he moved on to other topics,
it already had a user base, who kept it alive,
fitted it with the new SQL standard,
and eventually grew into the PostgreSQL development community we have today

1st movement

full-text search

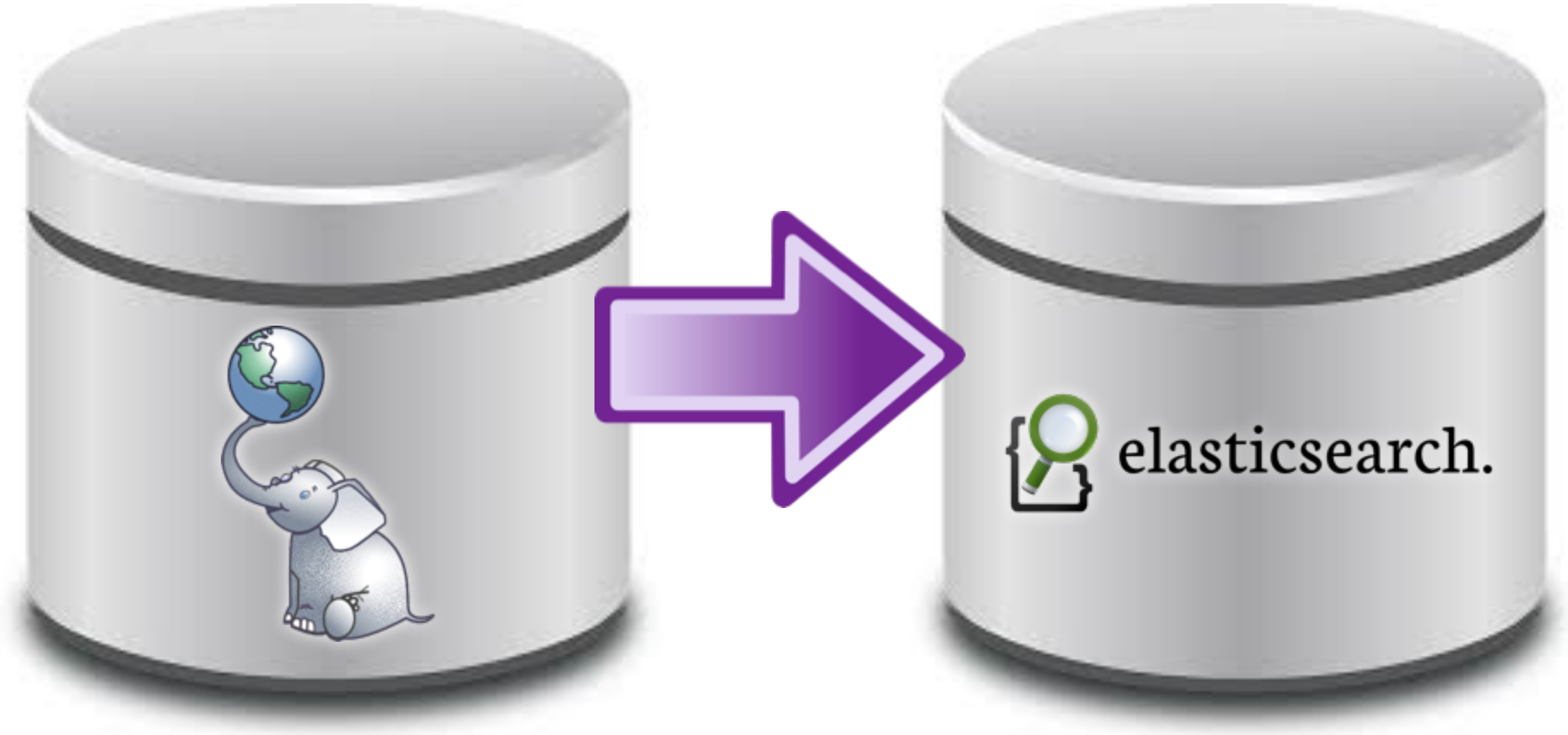
For our first movement I want to talk about
what’s possible when you start making use
of PostgreSQL’s native full-text search support.
Because if there’s a phrase that makes me want to put my head in the oven it’s

“we’re using PostGIS with elastic search”



“we’re using PostGIS with elastic search”
gah!
and I acknowledge, lucene, elastic search are nice tools,
but boy I sure hope you really need every scrap of functionality they offer,
because once you have two different data storage and query systems strapped together
everything in your system gets more complex and uglier

data synchronization



assuming, HOPEFULLY, that your **relational database** is your source of truth,
all the changes have to be replicated over to the elastic search system,
which adds a synchronization step to all work,
and if your data changes fast enough that can be quite complex,
but that’s actually the EASY problem

requires cross system **query**

elasticsearch.

the HARD problem is that once you have two query end points,
any query that involves BOTH a text search AND a spatial component
of **sufficient complexity** to require PostGIS
requires that the middleware starts to coordinate the query process

So it first talks to one of the systems and says
<x> give me all your records that match this text search query,
and then it has to take that information and walk over to the other system and say
<x> give me all the records you have that are in this set and fit my spatial clause.
And depending on the query, the order you want to do that in
(text first, or spatial first) varies:
basically you have to build a **little query planner in your middleware**.
What a TERRIBLE IDEA!

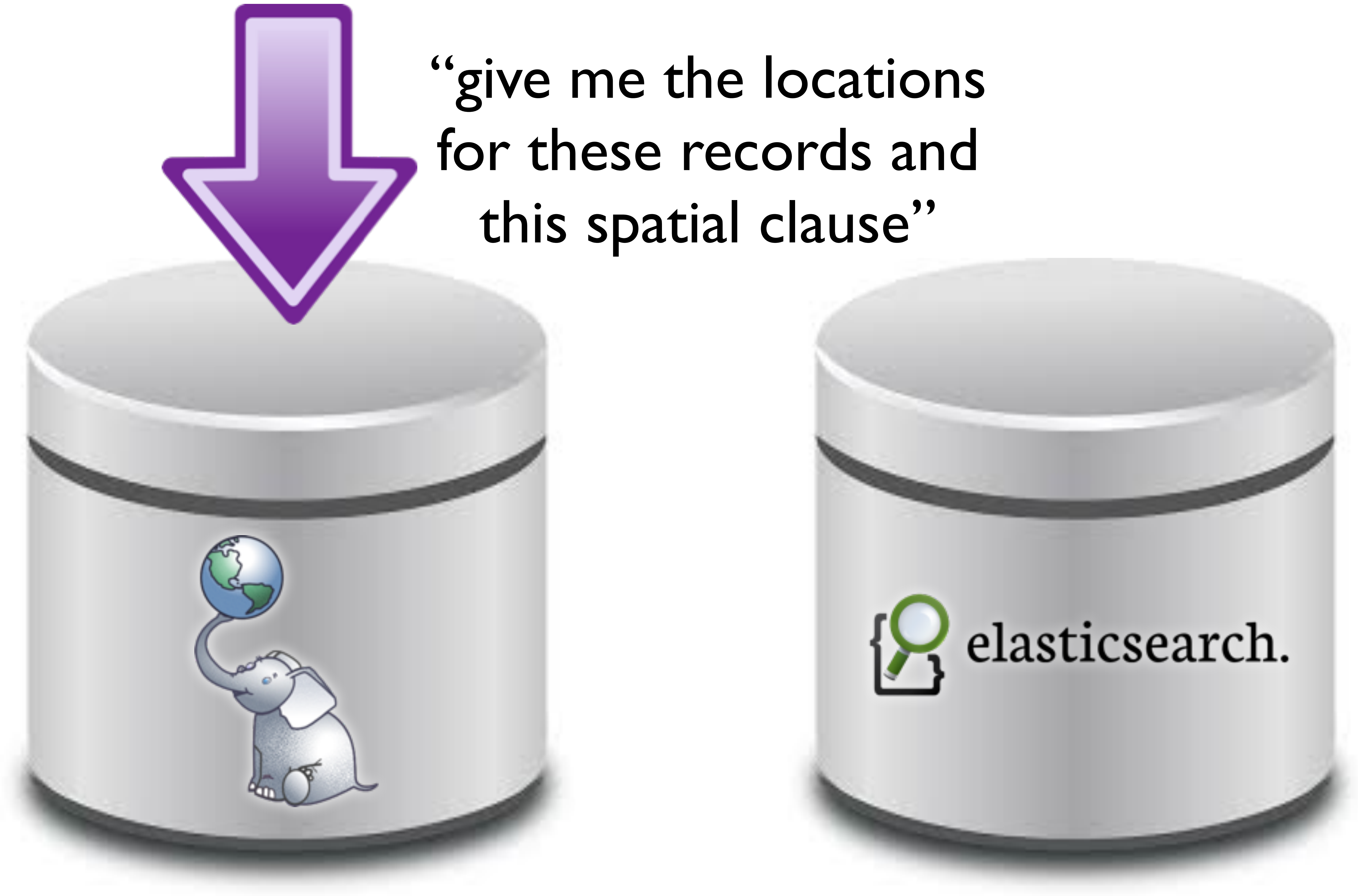
Particularly since PostgreSQL **actually has a full-text search system built into it.**

“give me the records for this term”

elasticsearch.

So it first talks to one of the systems and says
<x> give me all your records that match this text search query,
and then it has to take that information and walk over to the other system and say
<x> give me all the records you have that are in this set and fit my spatial clause.
And depending on the query, the order you want to do that in
(text first, or spatial first) varies:
basically you have to build a **little query planner in your middleware**.
What a TERRIBLE IDEA!

Particularly since PostgreSQL **actually has a full-text search system built into it.**



“give me the locations
for these records and
this spatial clause”

So it first talks to one of the systems and says
<x> give me all your records that match this text search query,
and then it has to take that information and walk over to the other system and say
<x> give me all the records you have that are in this set and fit my spatial clause.
And depending on the query, the order you want to do that in
(text first, or spatial first) varies:
basically you have to build a **little query planner in your middleware**.
What a TERRIBLE IDEA!

Particularly since PostgreSQL **actually has a full-text search system built into it.**

PostgreSQL TSearch

- full-text index
- language-aware stemming
- wild-card (prefix) searches
- weighted searches
- customizable synonym/stop dictionaries
- results ranking
- results highlighting

PostgreSQL Tsearch has all the basic capabilities you want in a full text search engine...
it has:
- stemming (foxes and fox, running and run)
- weighted searches (give more precedence to results matching document title)
- ability to create your own dictionaries
- ranking of results based on quality of match
- highlighting matched terms in output

but what does this have to do with magical **postgis**?
well, if your full-text engine and your spatial engine are in the same database,
you can run compound spatial/text queries and
not have to think about execution path or efficiency:
the database engine just DOES IT FOR YOU AUTOMATICALLY!

Geographic **N**ames **I**nformation **S**ystem

http://download.geonames.org/export/dump/US.zip

so, here is a a fun example application,
it is built using geographic names, in this case from geonames.org,
because geographic names are basically words,
really really short documents, that come with locations.

But any document type with location can be used to build a cool text/spatial location application.

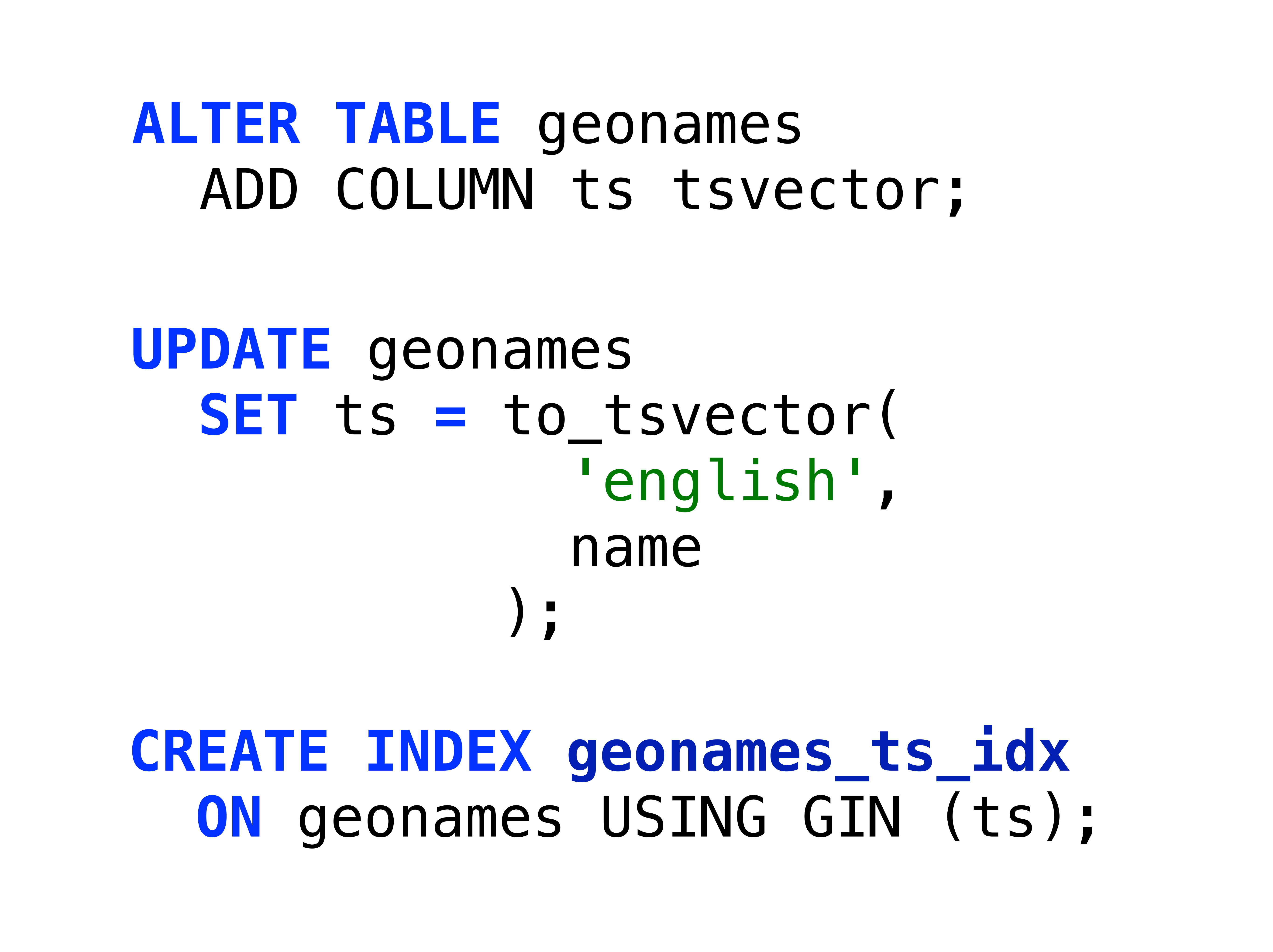
```
CREATE TABLE geonames
(
  geonameid INTEGER PRIMARY KEY,
  name VARCHAR(200),
  geog Geography(Point, 4326)
);
```

http://workshops.boundlessgeo.com/tutorial-wordmap/

With a little data mangling you can turn that geographic location names file into a table that looks like this.

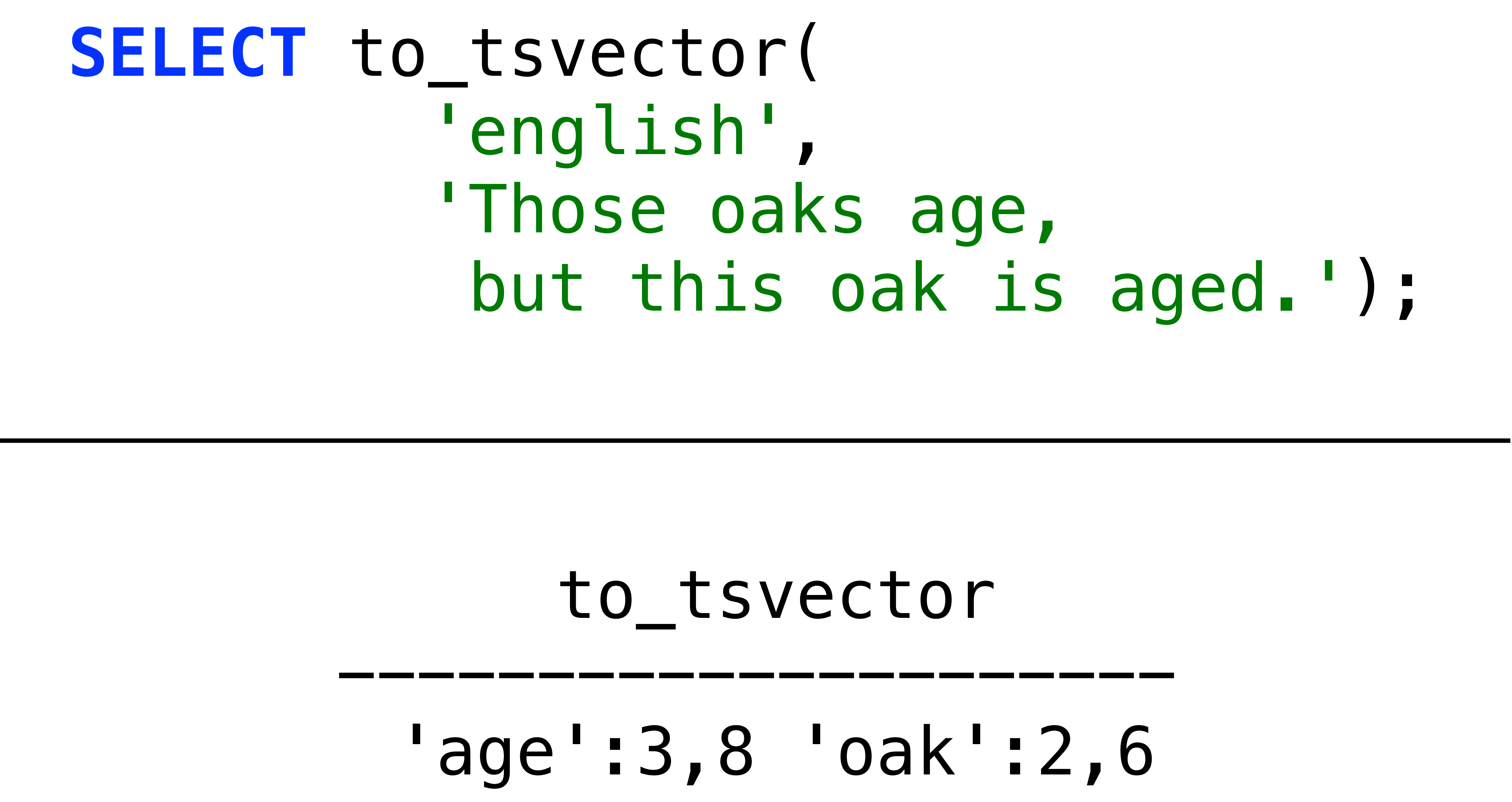
A primary key,
the name itself,
and a location

In order to get full-text searching enabled, you have to add a tsvector column



```
ALTER TABLE geonames
  ADD COLUMN ts tsvector;

UPDATE geonames
  SET ts = to_tsvector(
             'english',
             name
           );

CREATE INDEX geonames_ts_idx
  ON geonames USING GIN (ts);
```

- so our column type for full text searching is ‘tsvector’
- then we populate it with ‘tsvector’ data, using an ‘english’ configuration, more about that later
- and finally we index, using the fulltext index for ‘tsvector’:
a ‘GIN’ or “generalized inverted index”,
which is also used in PostgreSQL index support for array types



```
SELECT to_tsvector(
          'english',
          'Those oaks age,
           but this oak is aged.');
```

```
      to_tsvector
---------------------
 'age':3,8 'oak':2,6
```

Note, there is a “magic parameter” in here, the word “english”

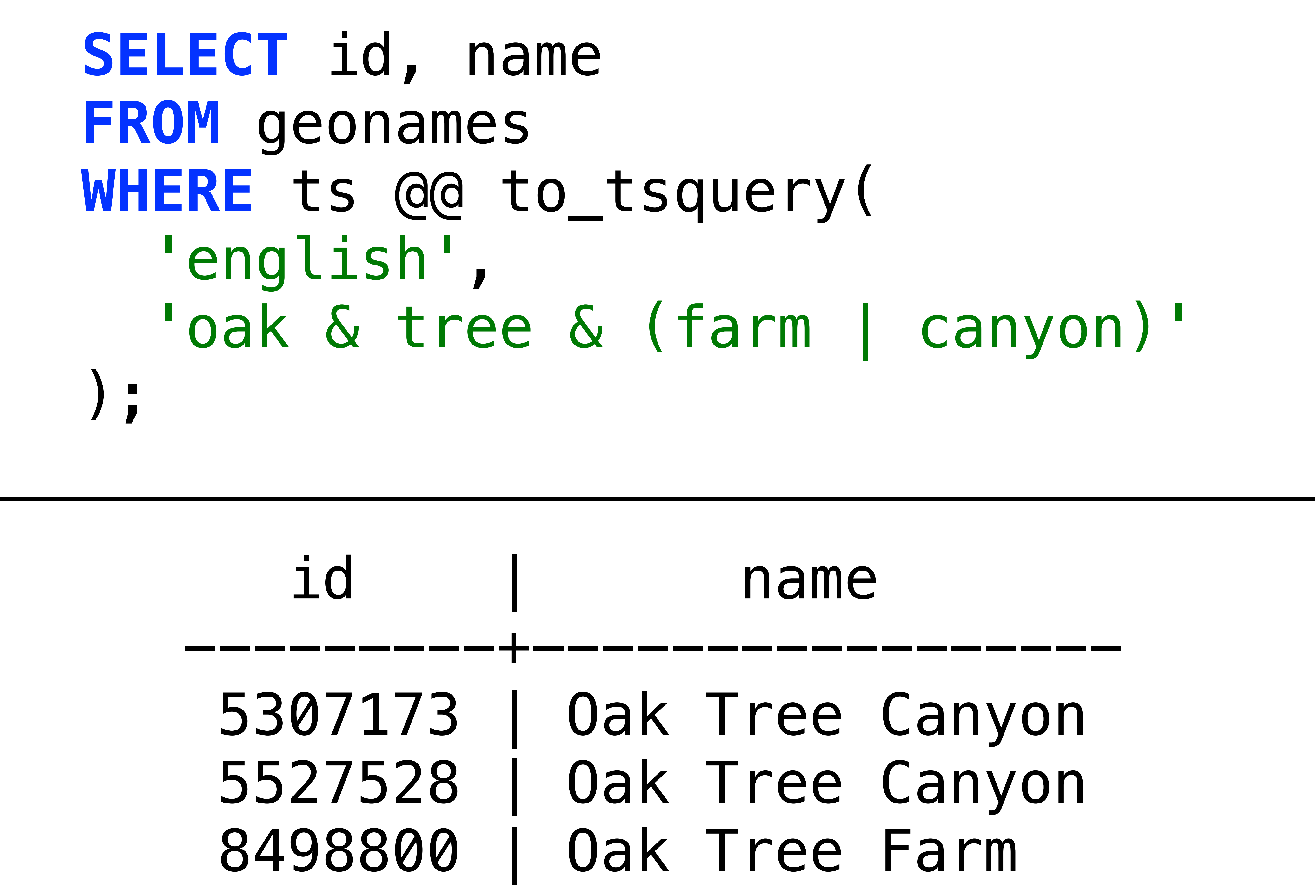
we’ve specified an english configuration,
so english grammar rules are used to determine that “oak” and “oaks” and “age” and “aged” are basically the same thing, to identify all the articles and pronouns that can be ignored and reduce the phrase to a simple vector-like structure suitable for indexing.

SO, to_tsvector gives us a column of “tsvectors” we can query, but, HOW do we do that?

```sql
SELECT to_tsquery(
          'english',
          'oak & (tree | ridge)');
```

```
        to_tsquery
----------------------
'oak' & ( 'tree' | 'ridge' )
```

To query a tsvector you need a tsquery, which is itself a logical filter
You can construct one as a combination of “and” and “or” clauses,
optionally with weighting and partial matching,
this is a query that would match entries
with both “oak and tree” or “oak and ridge”



```
SELECT id, name
FROM geonames
WHERE ts @@ to_tsquery(
  'english',
  'oak & tree & (farm | canyon)'
);
```

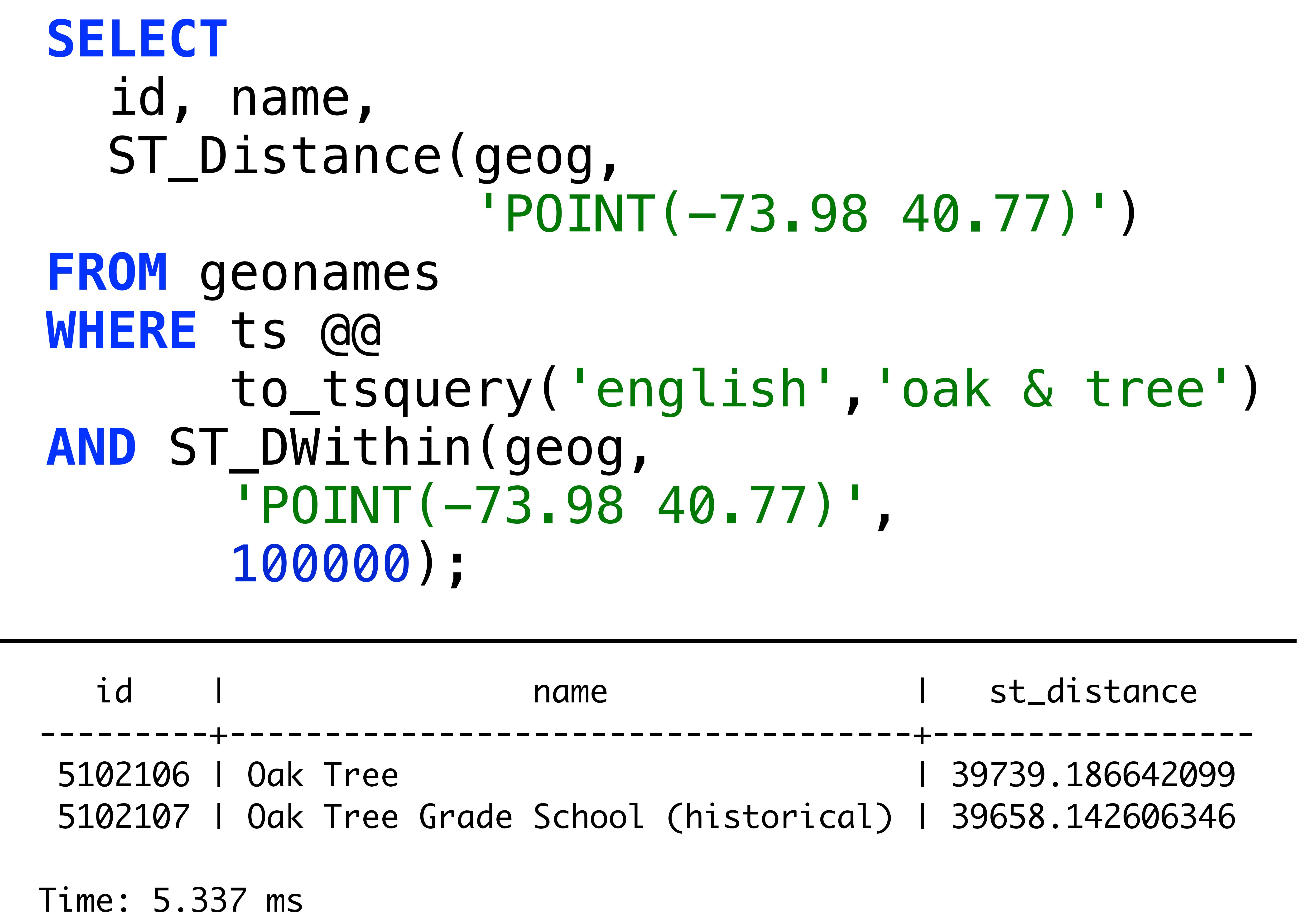
```
   id    |       name
---------+-----------------
 5307173 | Oak Tree Canyon
 5527528 | Oak Tree Canyon
 8498800 | Oak Tree Farm
```

and we can use the query in a full text search of our
2M record geographic names table using the @@ operator
to find all the tsvectors that match the tsquery.
It turns out there are only three,
but the REALLY REALLY interesting thing is how quickly it finds the answer

from
2200722 rows

17.598 ms

just 17ms! that’s a good fast search of 2.2M records
and the best part is that, now that the full-text search is handled **inside the database**,
it’s possible to build **efficient compound spatial/text queries too**



```
SELECT
  id, name,
  ST_Distance(geog,
             'POINT(-73.98 40.77)')
FROM geonames
WHERE ts @@
      to_tsquery('english','oak & tree')
AND ST_DWithin(geog,
      'POINT(-73.98 40.77)',
      100000);
```

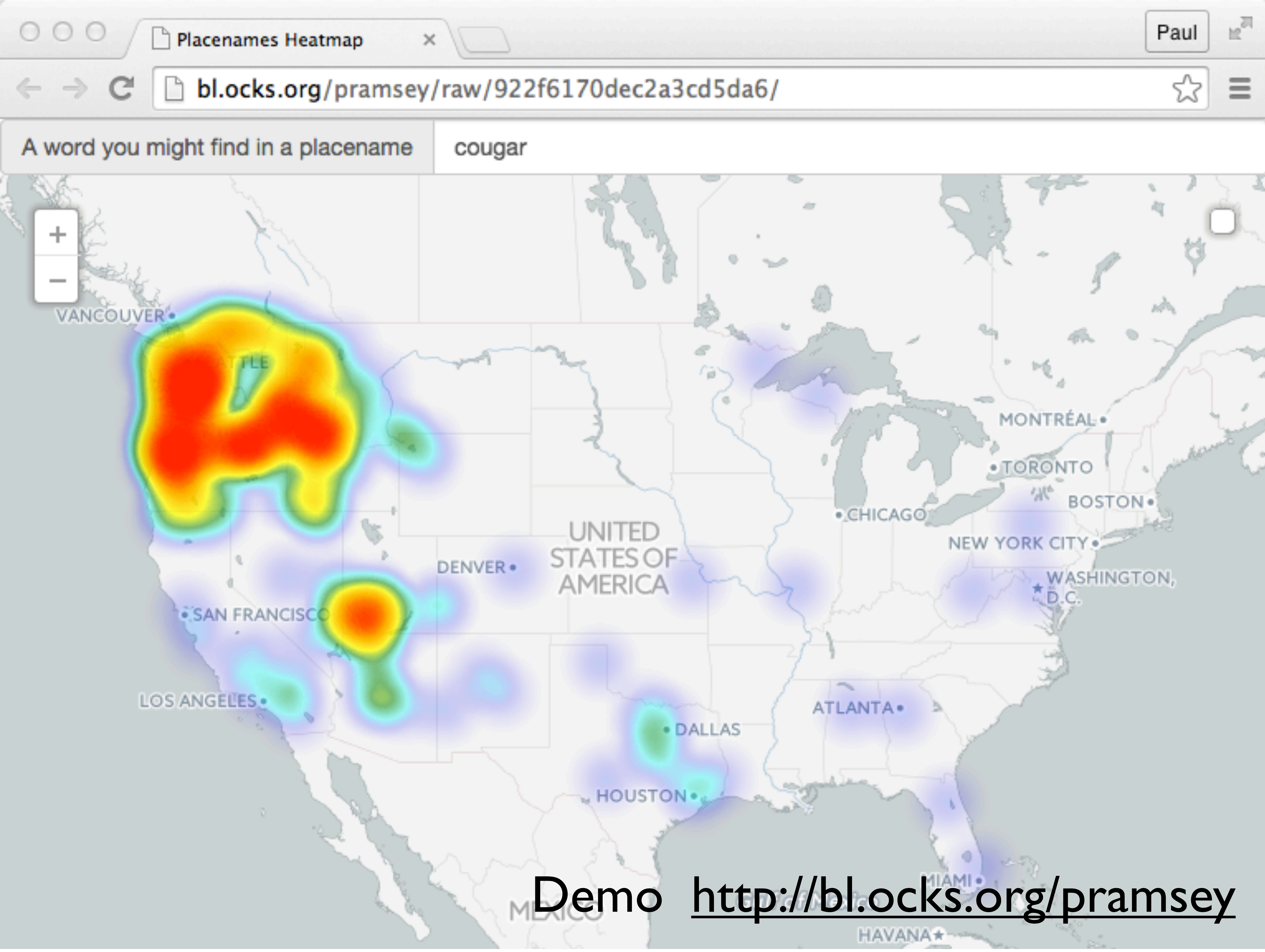
```
   id    |                 name                  |   st_distance
---------+---------------------------------------+-----------------
 5102106 | Oak Tree                              | 39739.186642099
 5102107 | Oak Tree Grade School (historical)    | 39658.142606346

Time: 5.337 ms
```

Like this query, which **combines** a search for all records with oak and tree
with a spatial filter restricting the result to the nearest 100km.
And because both clauses are handled by the database,
all the database machinery is at your disposal,
figuring out the most efficient way to access the rows

```
Bitmap Heap Scan on geonames
 (actual time=3.698..3.763 rows=2 loops=1)
   Recheck Cond:
    (ts @@ '''oak'' & ''tree'''::tsquery)
   Filter: _st_dwithin(...)
   Rows Removed by Filter: 57
   ->  Bitmap Index Scan on geonames_ts_idx
       (actual time=3.310..3.310 rows=59 loops
         Index Cond:
          (ts @@ '''oak'' & ''tree'''::tsquery
```

This is the explain analyze output for the last query, and reading from the bottom up, you can see that, in this case, the database ran the full-text search first, because it was the most selective
(only returning 59 rows, as opposed to the “all things w/i 100km filter, which would have returned several thousand).
Then it applied the spatial filter, which removed 57 of the 59, leaving just the 2 we got in the result set.

OK, so I’ve shown a lot of SQL and maybe now you’re starting to wonder where the MAGIC is.
Usually the MAGIC comes when you
bind the power of SQL into a user interface
and make that power visually manifest.

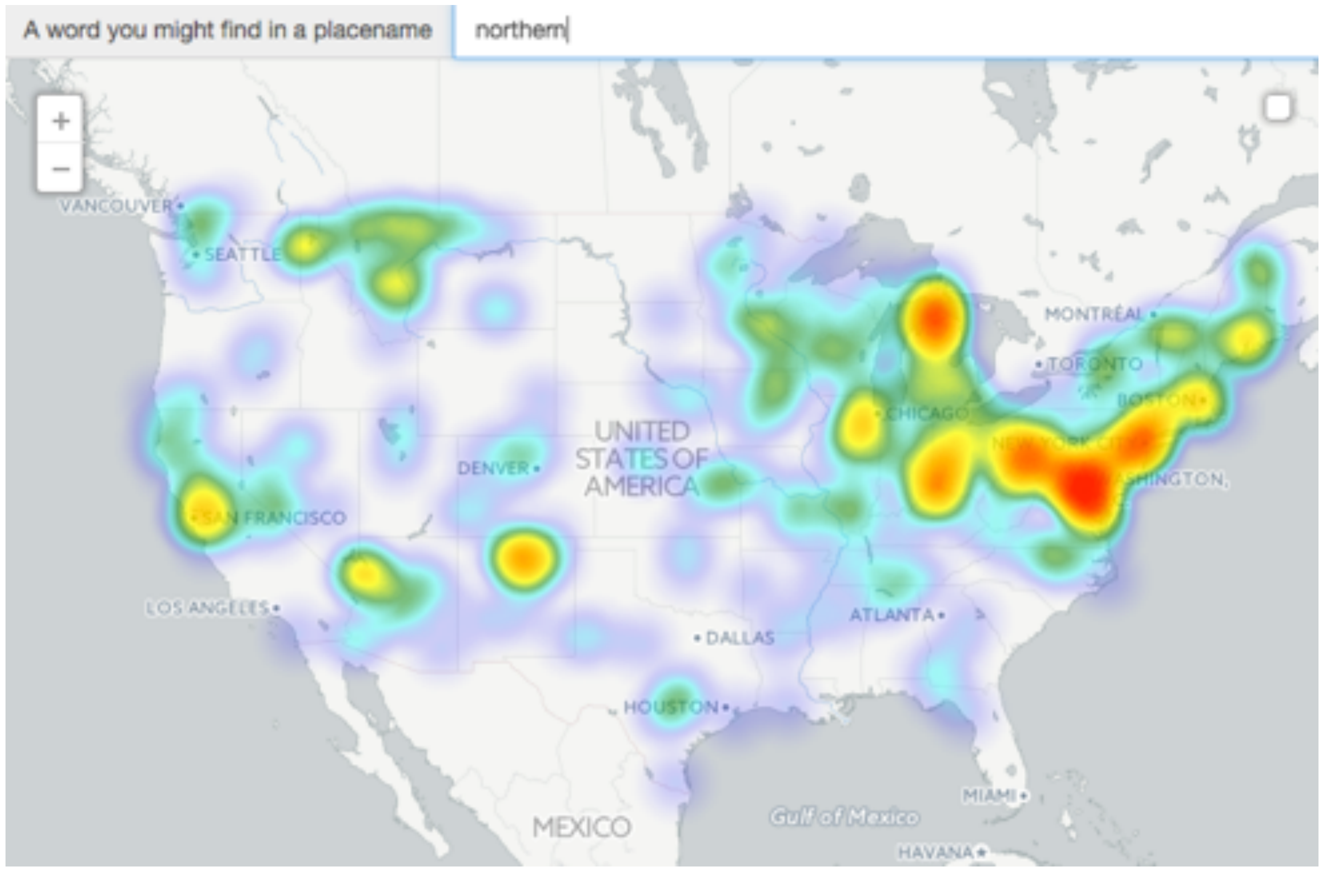


A word you might find in a placename | cougar

Demo http://bl.ocks.org/pramsey

take all those place names, and subset them quickly using text search
and pass the result into a heat map!
here's the map for our own REGIONALISM in the pacific northwest:
in Cascadia, we call "mountain lions" "cougars"

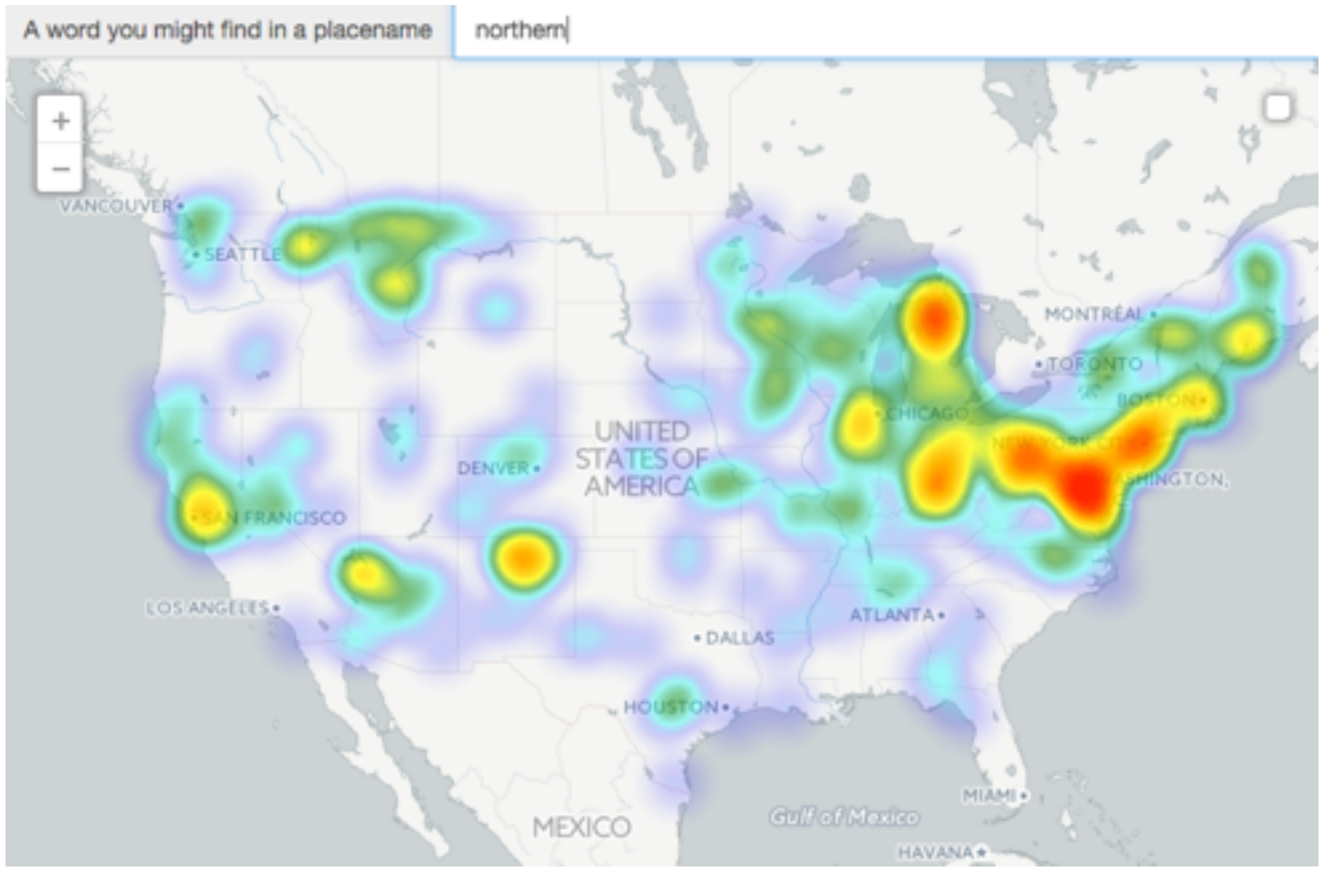
northern



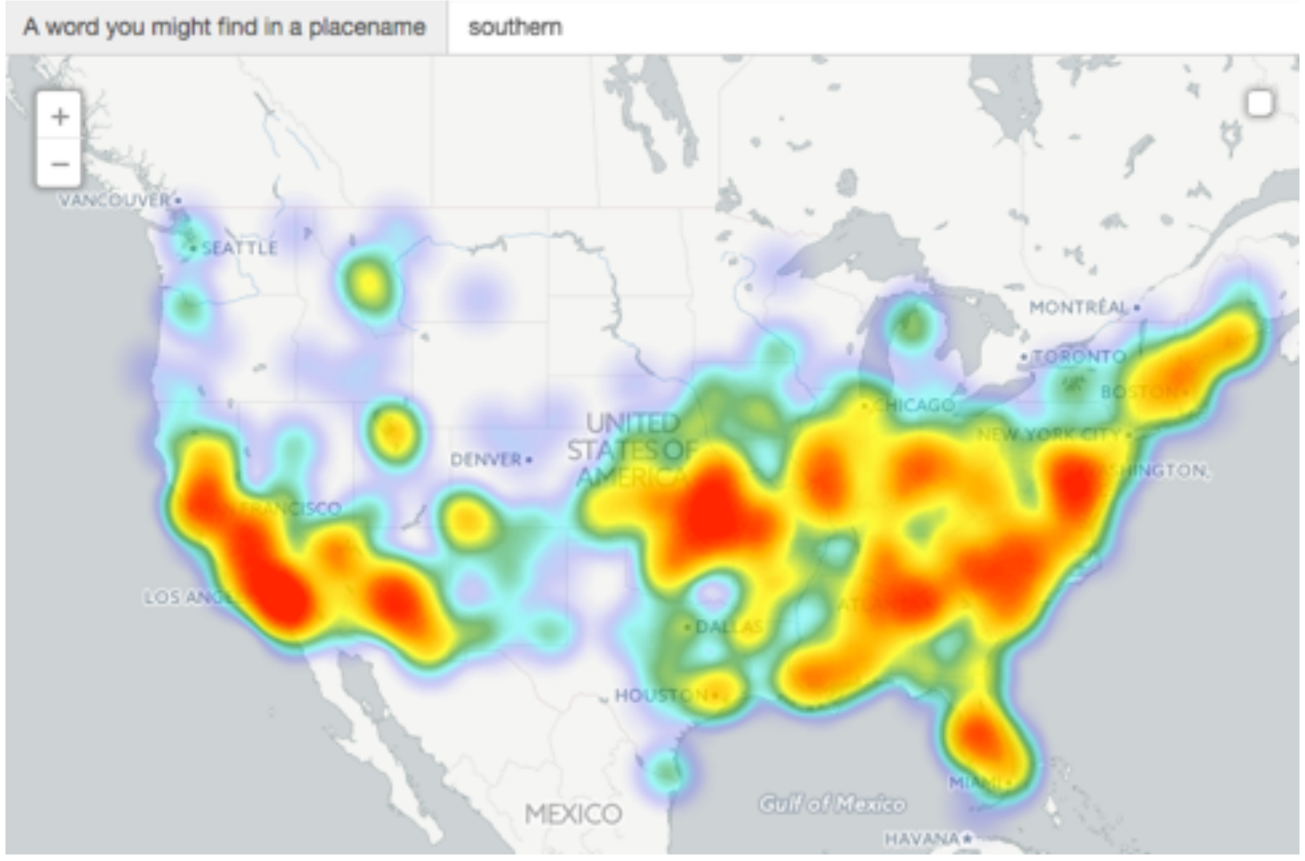
there’s all kinds of oddities on how we name things, and thus, how we perceive ourselves.
<x> <x> <x>

there’s obviously some extra cachet in being “western”

northern


A word you might find in a placename: northern

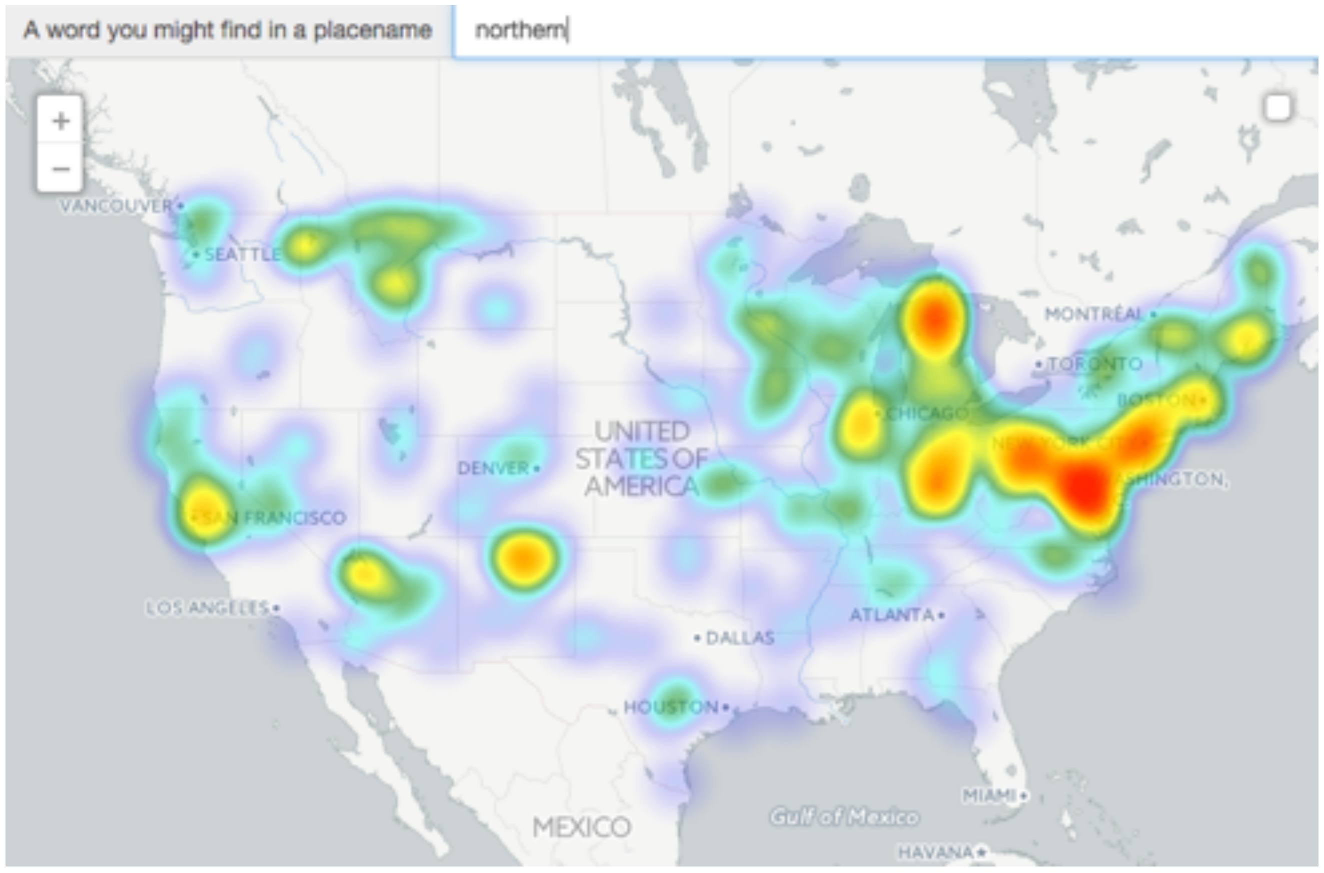
southern


A word you might find in a placename: southern

there’s all kinds of oddities on how we name things, and thus, how we perceive ourselves.
<x> <x> <x>

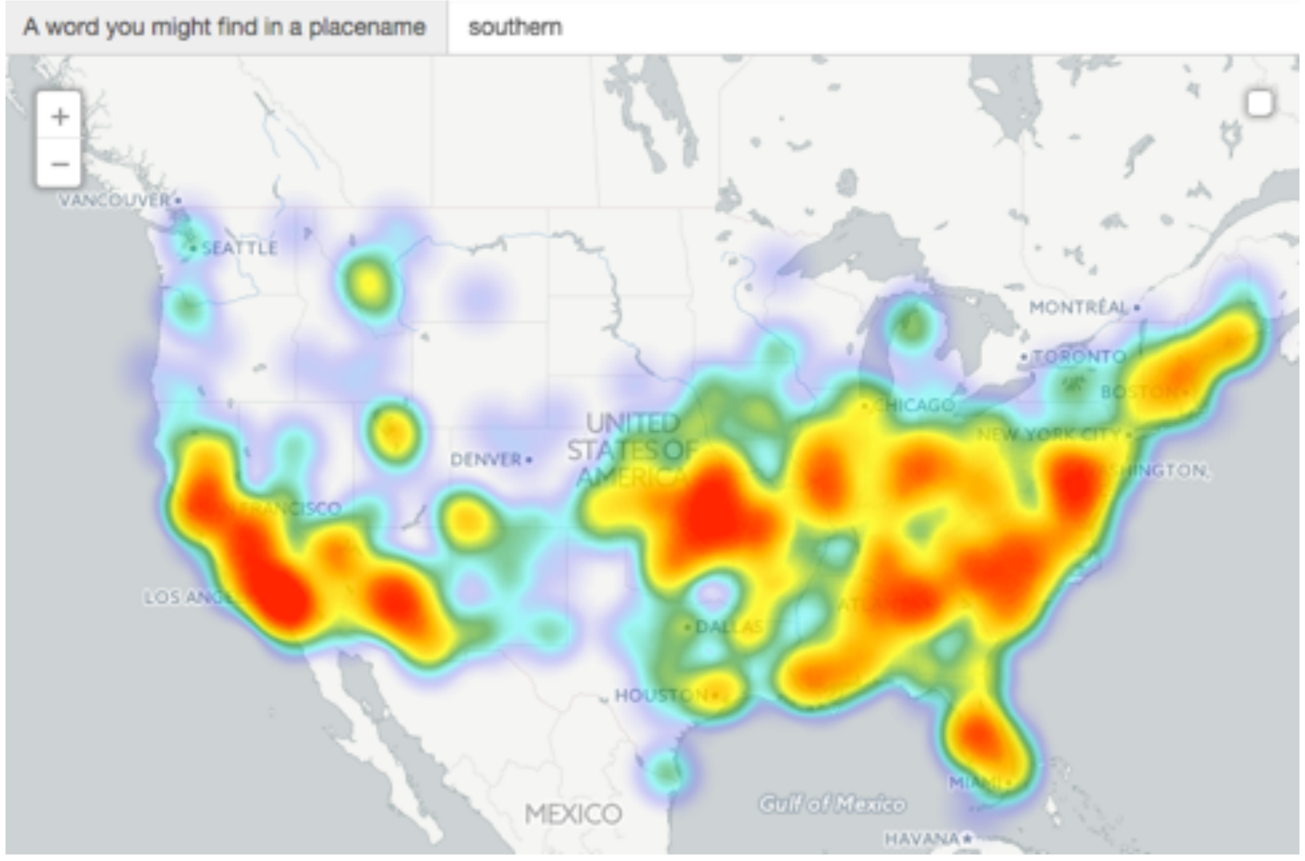
there’s obviously some extra cachet in being “western”

northern



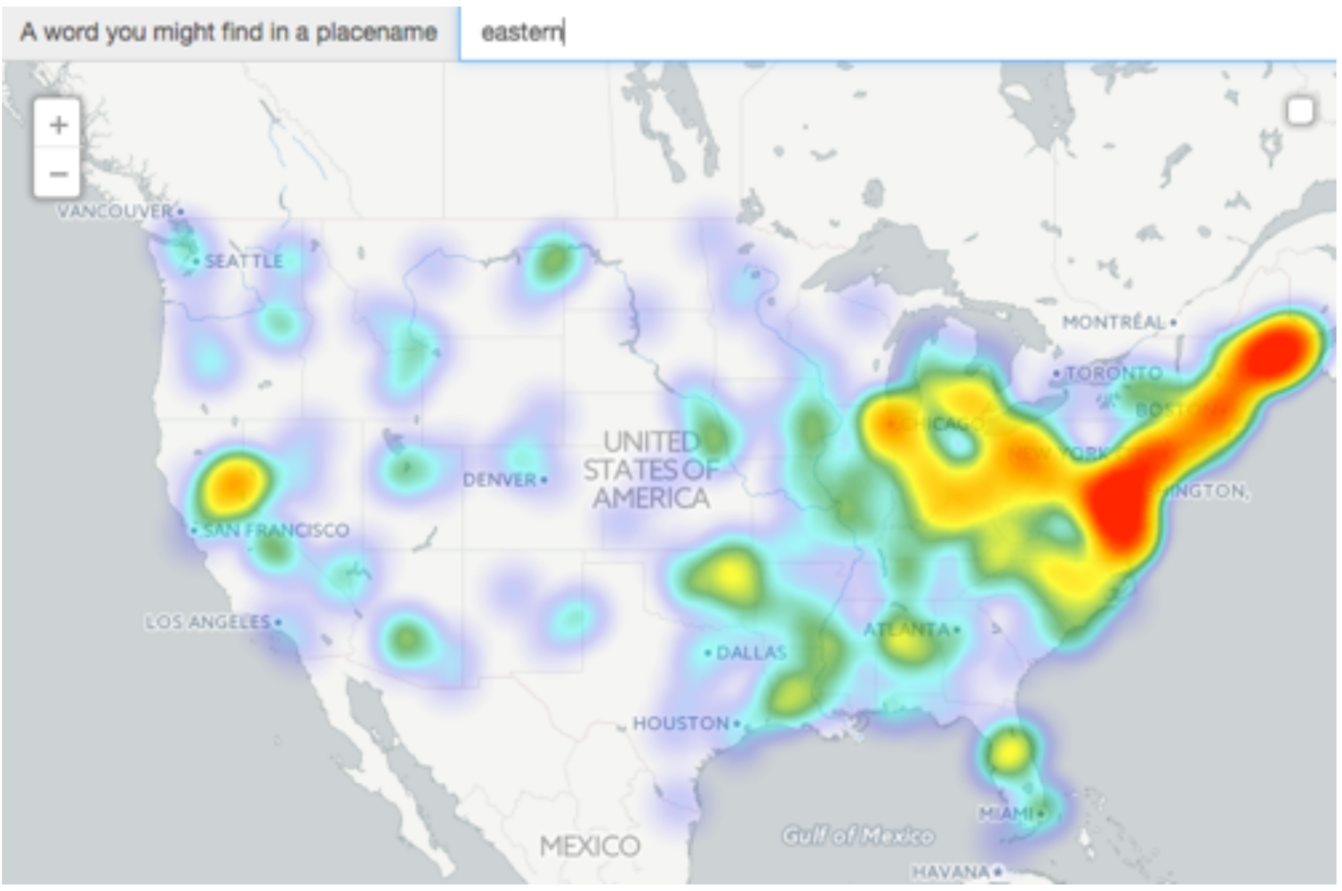
A word you might find in a placename northern

southern



A word you might find in a placename southern

eastern

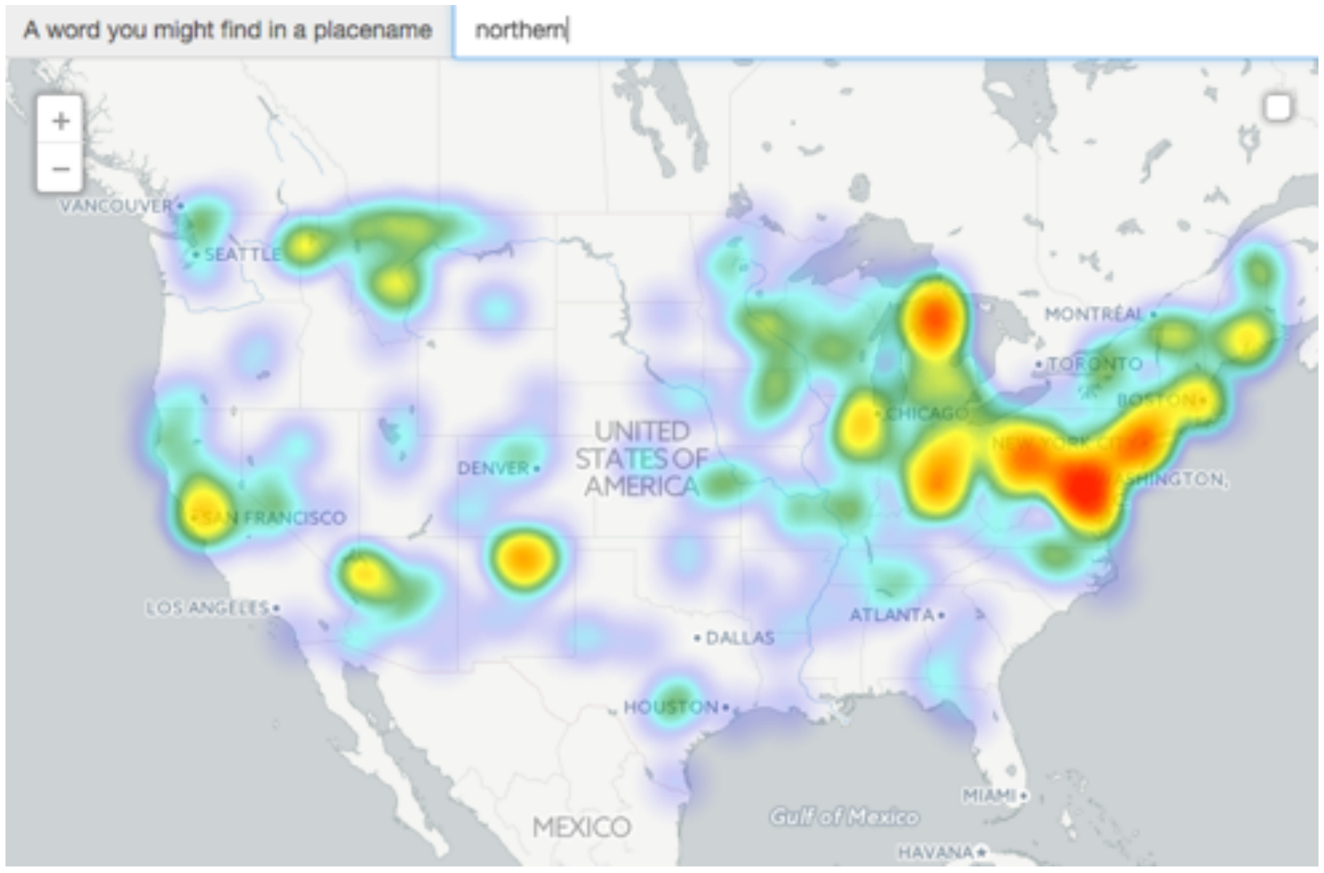


A word you might find in a placename eastern

there’s all kinds of oddities on how we name things, and thus, how we perceive ourselves.
<x> <x> <x>

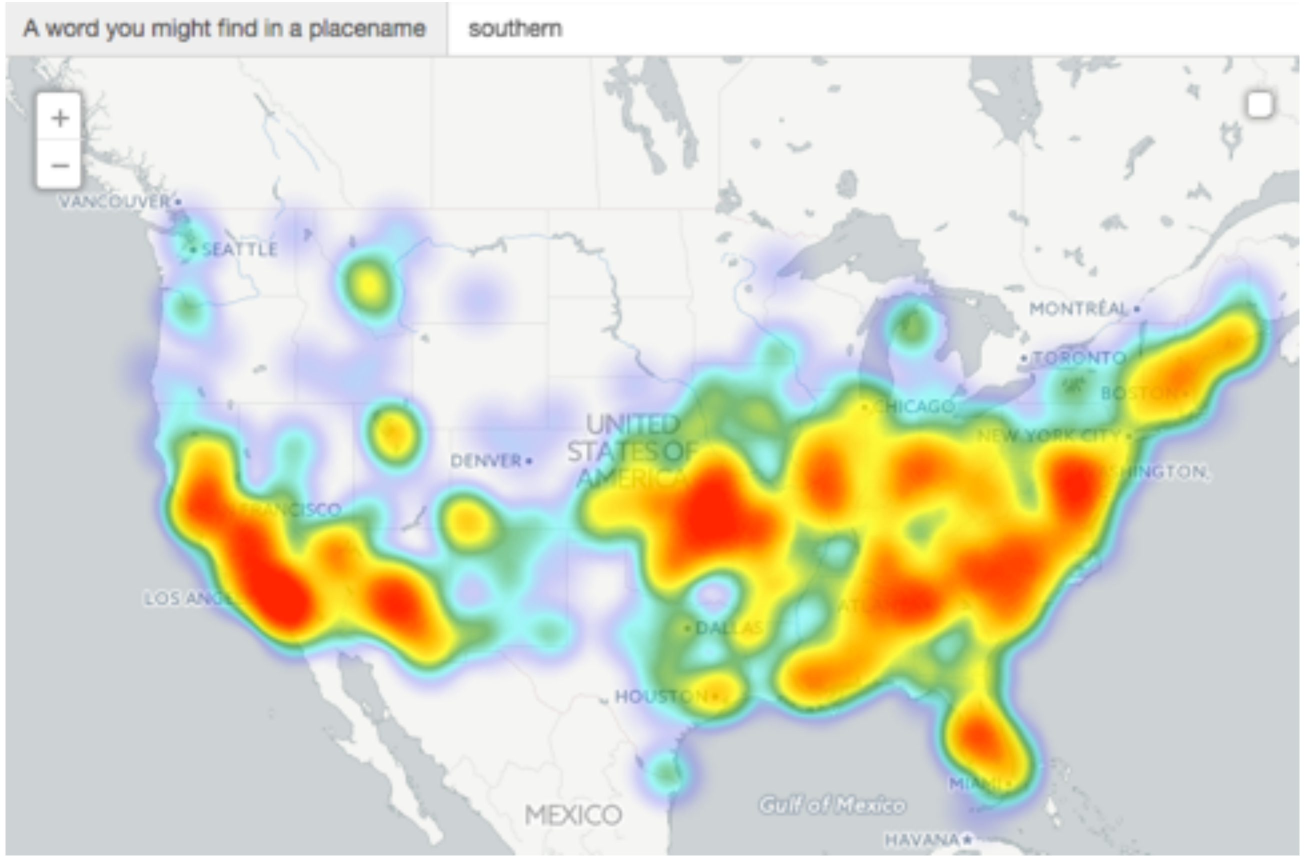
there’s obviously some extra cachet in being “western”

northern



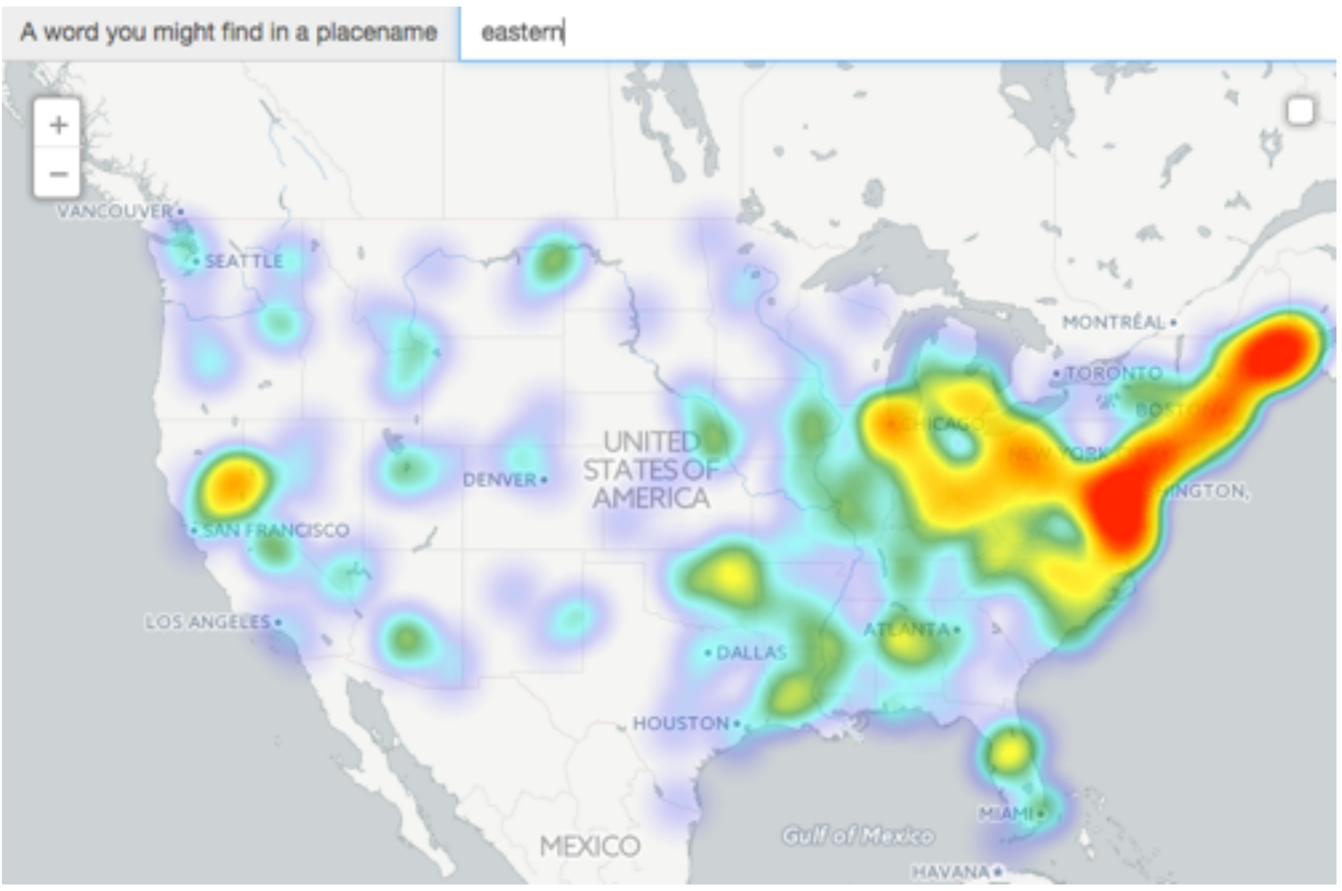
A word you might find in a placename: northern

southern



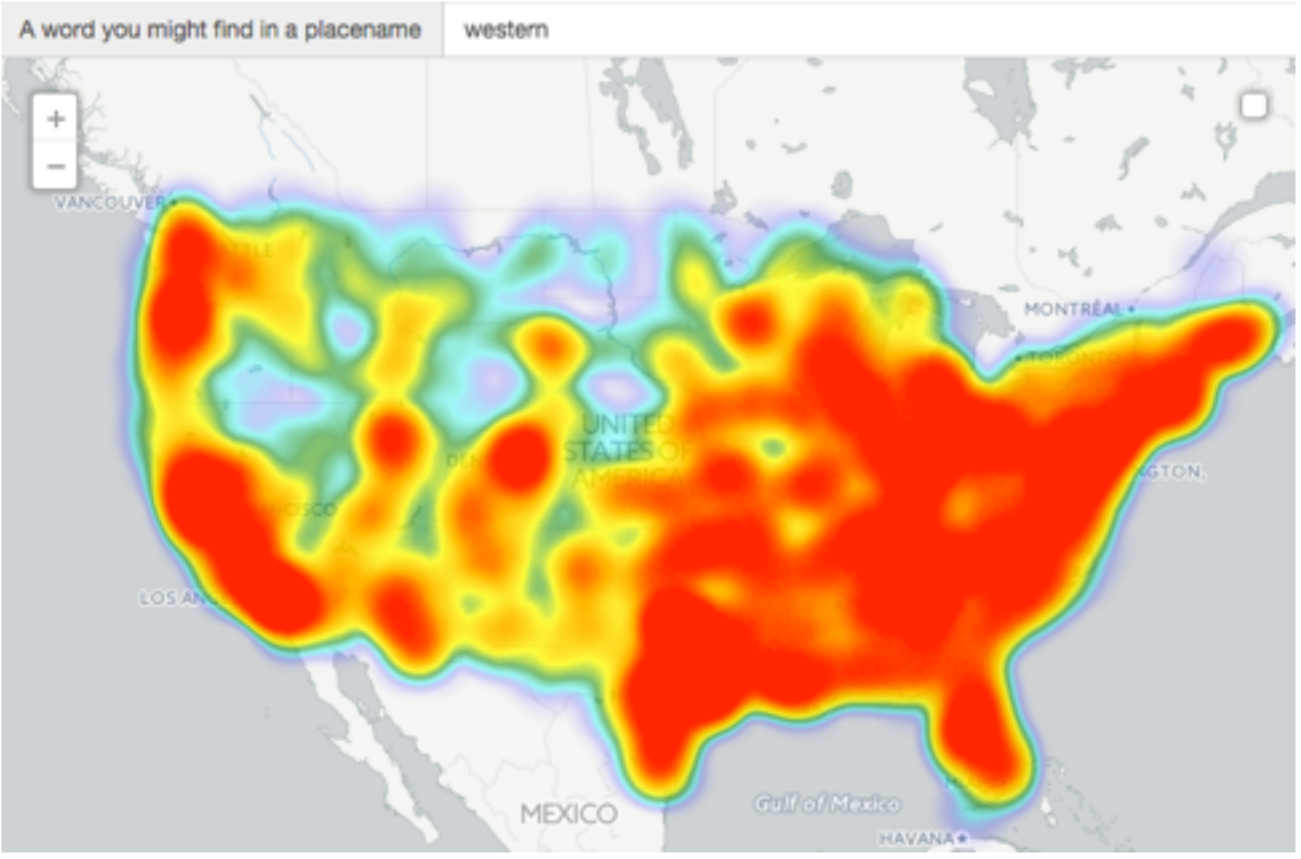
A word you might find in a placename: southern

eastern



A word you might find in a placename: eastern

western

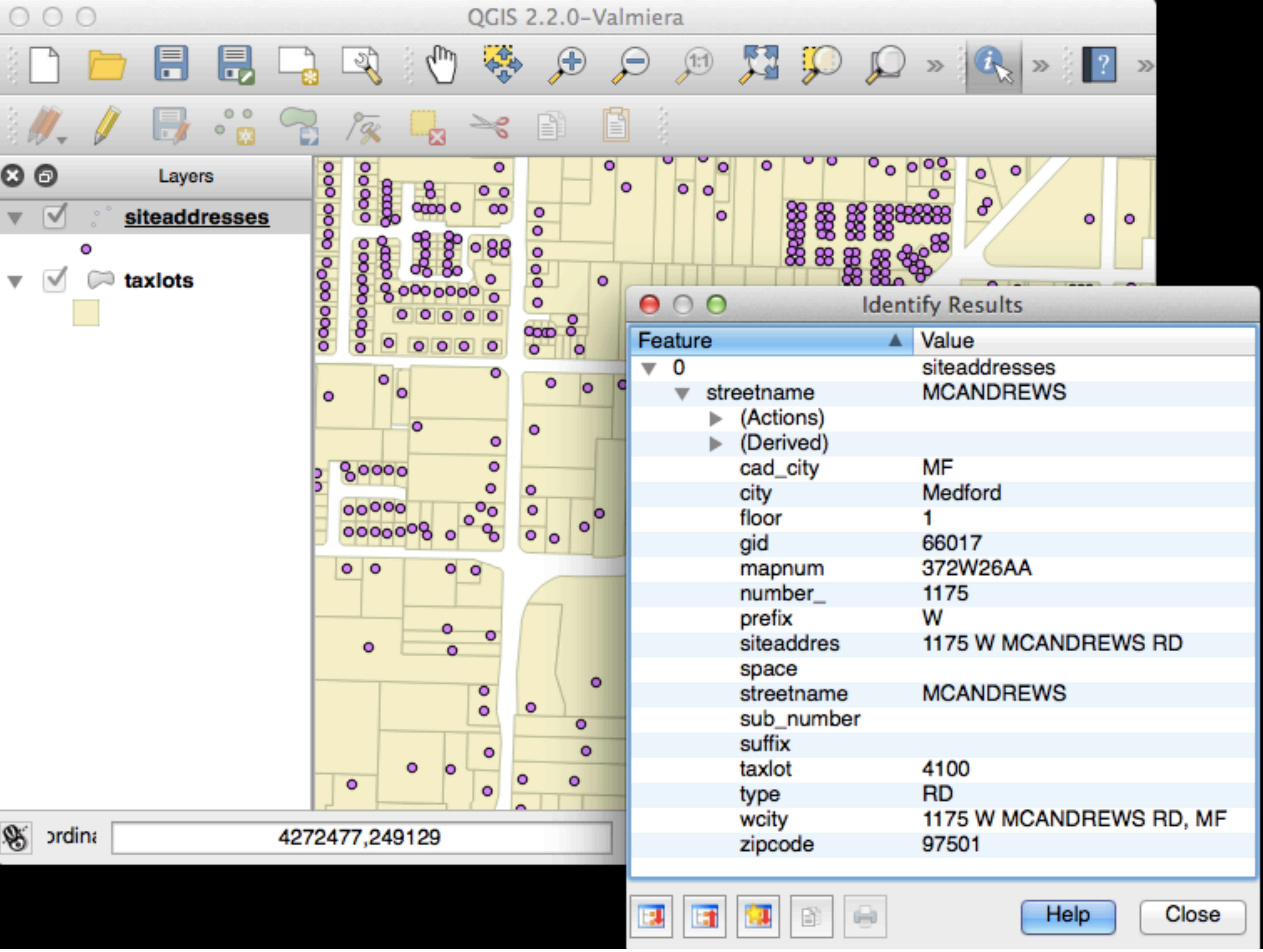


A word you might find in a placename: western

there’s all kinds of oddities on how we name things, and thus, how we perceive ourselves.
<x> <x> <x>

there’s obviously some extra cachet in being “western”

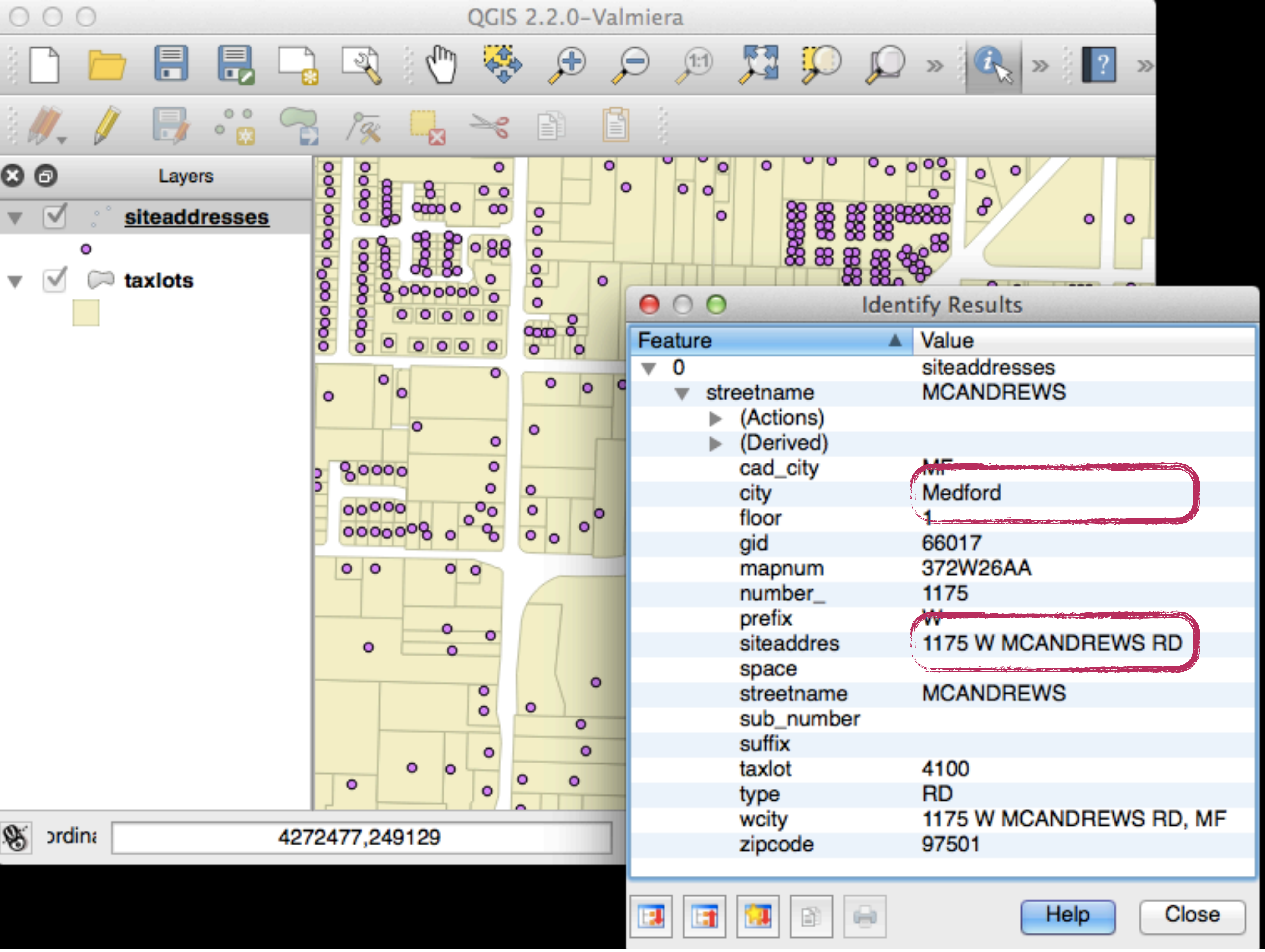
OK, that was magical, but perhaps not practical enough?
How about this one....



ordina 4272477,249129

Feature	Value
0	siteaddresses
streetname	MCANDREWS
(Actions)	
(Derived)	
cad_city	MF
city	Medford
floor	1
gid	66017
mapnum	372W26AA
number_	1175
prefix	W
siteaddres	1175 W MCANDREWS RD
space	
streetname	MCANDREWS
sub_number	
suffix	
taxlot	4100
type	RD
wcity	1175 W MCANDREWS RD, MF
zipcode	97501

Suppose you were a county with some standard parcel and address data,
and you wanted to set up a simple “parcel finder” application for folks to look up their home information.
How might you do it?
You want to do a “google style” interface, with just one input field and magical autocomplete...
Well, your GIS data has an street name, address number, and city for every site address!
Make use of that! But how?



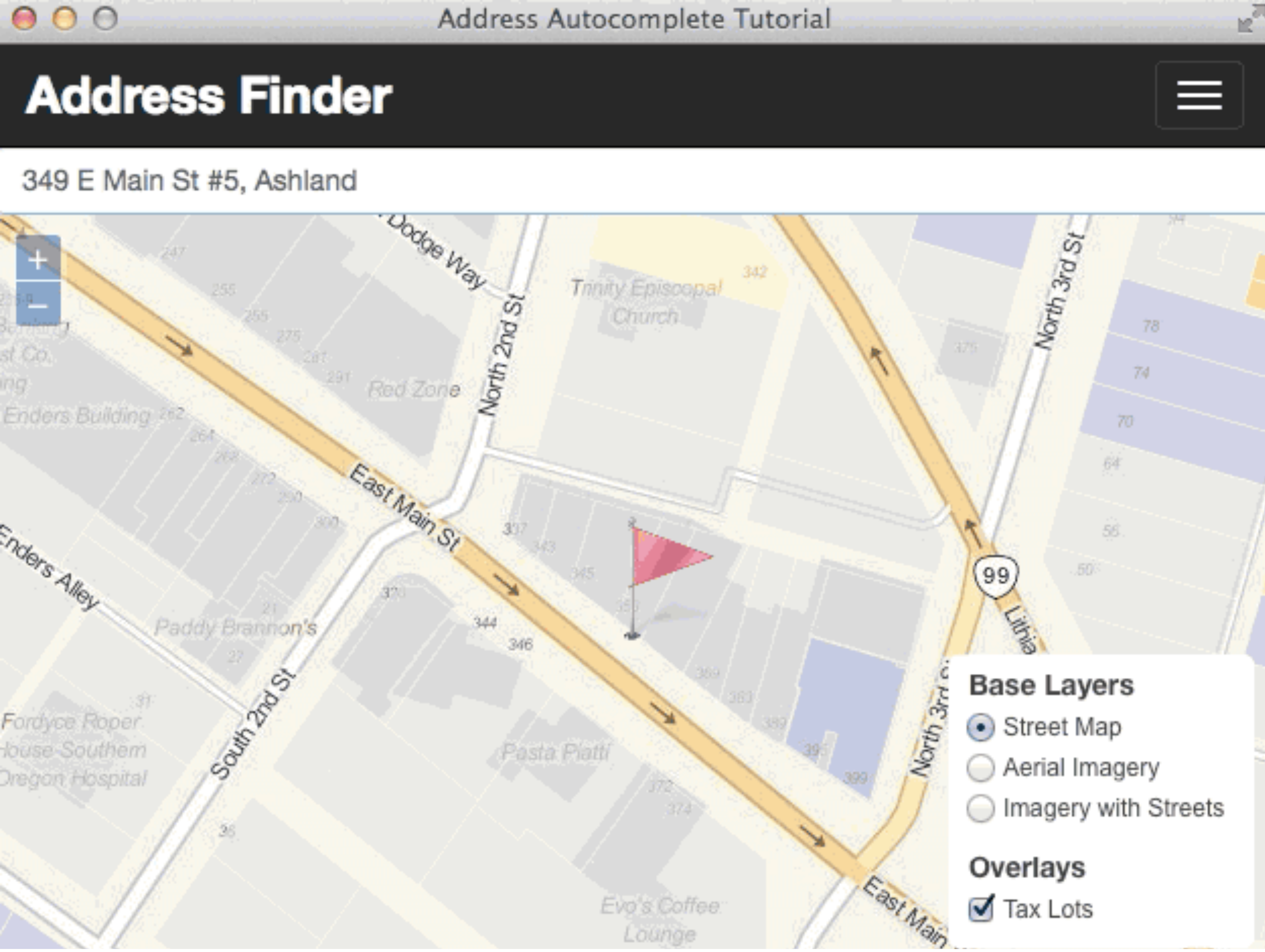
Feature	Value
0	siteaddresses
streetname	MCANDREWS
(Actions)	
(Derived)	
cad_city	MF
city	Medford
floor	1
gid	66017
mapnum	372W26AA
number_	1175
prefix	W
siteaddres	1175 W MCANDREWS RD
space	
streetname	MCANDREWS
sub_number	
suffix	
taxlot	4100
type	RD
wcity	1175 W MCANDREWS RD, MF
zipcode	97501

Suppose you were a county with some standard parcel and address data,
and you wanted to set up a simple “parcel finder” application for folks to look up their home information.
How might you do it?
You want to do a “google style” interface, with just one input field and magical autocomplete...
Well, your GIS data has an street name, address number, and city for every site address!
Make use of that! But how?

```
SELECT to_tsquery(
           'simple',
           '256 & mai:*');
```

```
        to_tsquery
----------------------
   '256' & 'mai':*
```

The only trick is that, to do an autocomplete form you have to be able to look up
not just the words the user has ALREADY ENTERED,
but also the work they are in the MIDDLE OF TYPING.
And fortunately PgSQL text search can DO THAT TOO!
See in the to_tsquery function, I'm not just looking for '256'
I'm also looking for "words that start with 'm-a-i'"
PostgreSQL text search calls this "prefix matching"



Address Autocomplete Tutorial

Address Finder

349 E Main St #5, Ashland

Base Layers

● Street Map

○ Aerial Imagery

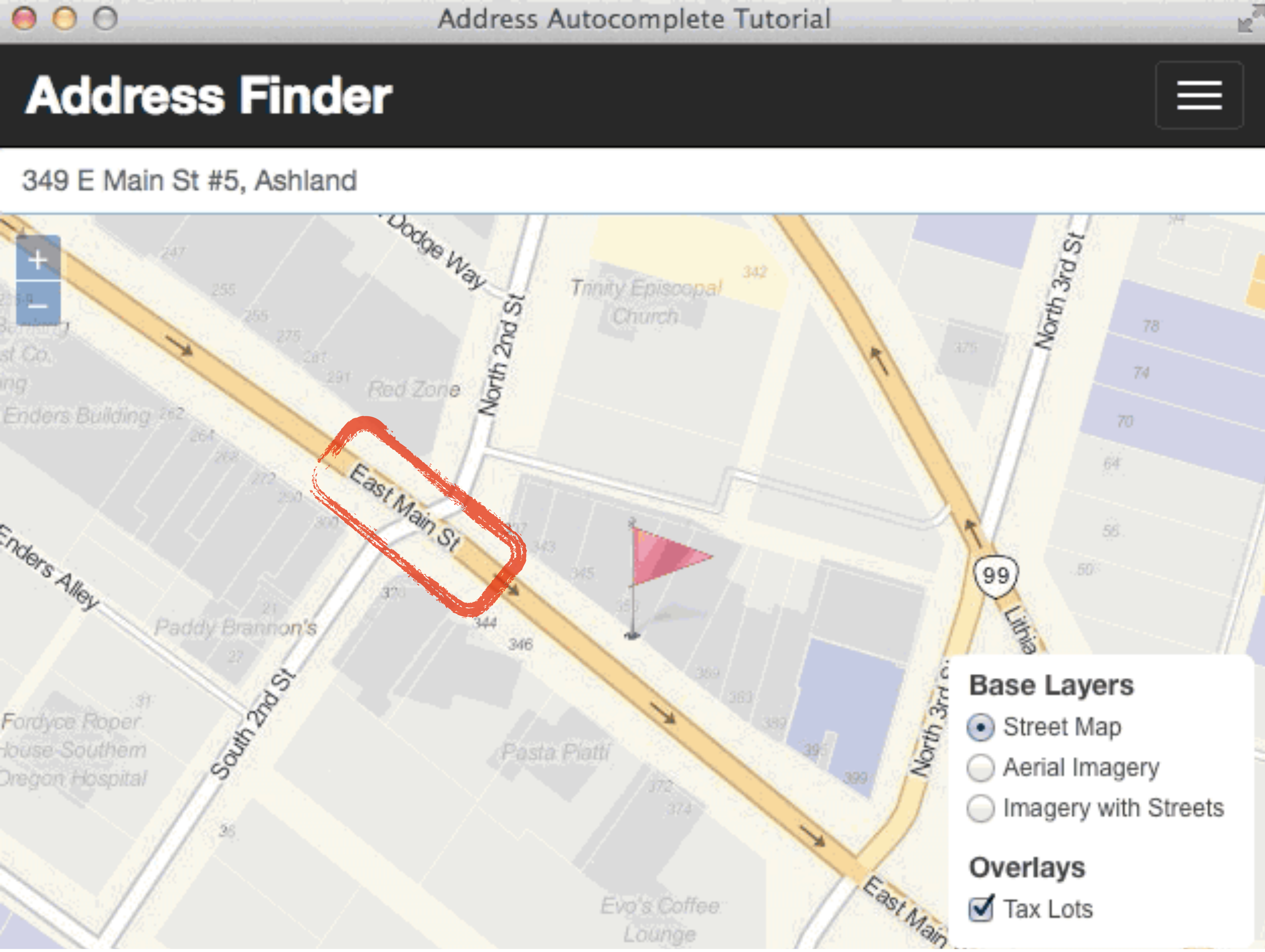
○ Imagery with Streets

Overlays

☑ Tax Lots

With prefix matching and a simple javascript (jquery UI, in this case) autocomplete form,
you can have a really fast autocomplete search box up and running in a few minutes.
And it's uncannily accurate. It doesn't care about word order.
If you want to get fancy, in addition having one row for each street address,
you can also add rows to your table for each street intersection, like "main and second"

This last example here is interesting, because in the search field, we've got
349 "E" main "st" and on the map (a google base map in this case)
we have "EAST" (all spelled out) "main st".



Address Autocomplete Tutorial

Address Finder

349 E Main St #5, Ashland

Base Layers

● Street Map

○ Aerial Imagery

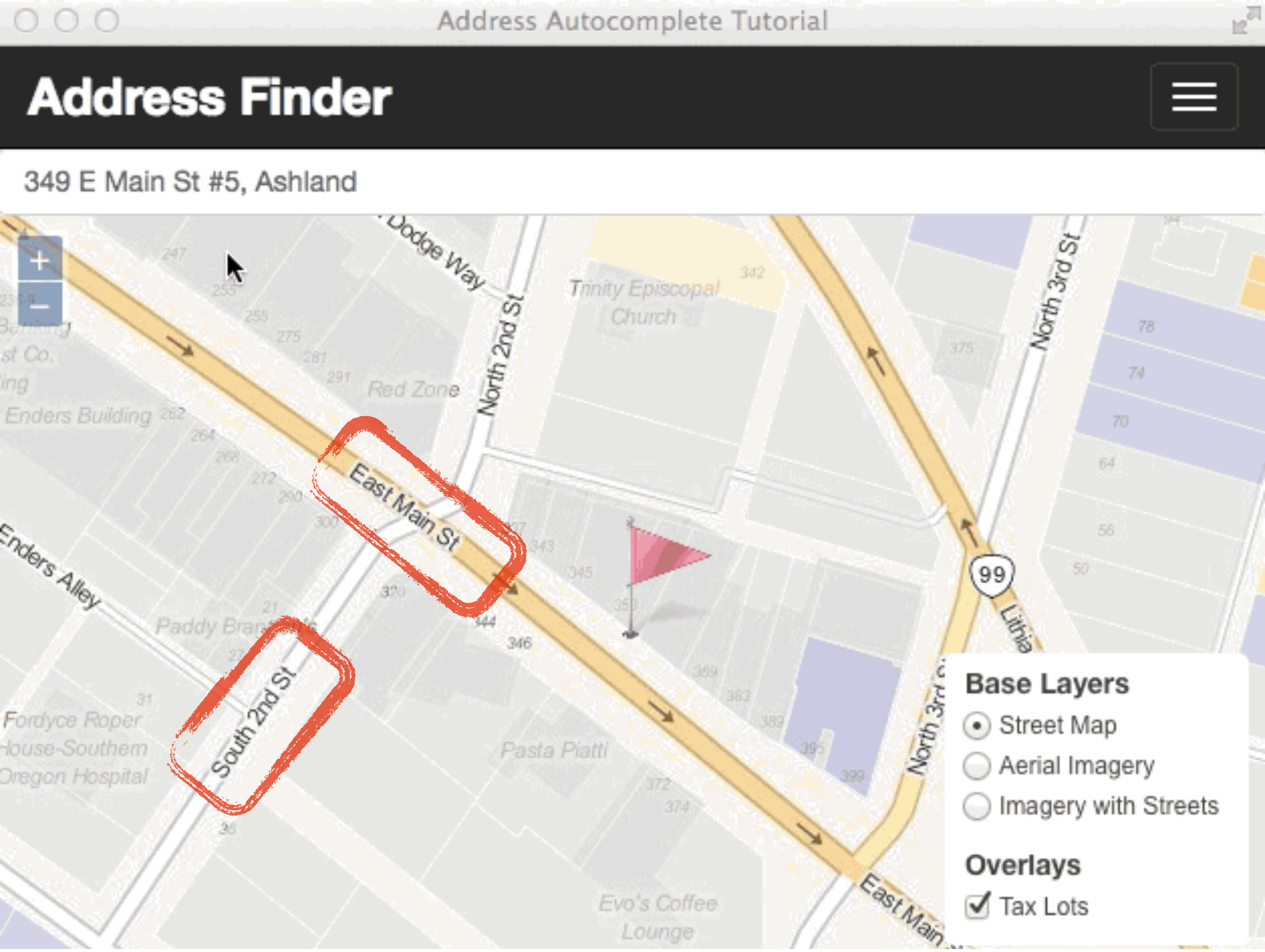
○ Imagery with Streets

Overlays

☑ Tax Lots

With prefix matching and a simple javascript (jquery UI, in this case) autocomplete form,
you can have a really fast autocomplete search box up and running in a few minutes.
And it’s uncannily accurate. It doesn’t care about word order.
If you want to get fancy, in addition having one row for each street address,
you can also add rows to your table for each street intersection, like “main and second”

This last example here is interesting, because in the search field, we’ve got
349 “E” main “st” and on the map (a google base map in this case)
we have “EAST” (all spelled out) “main st”.


Address Autocomplete Tutorial

Address Finder

349 E Main St #5, Ashland

Base Layers

● Street Map

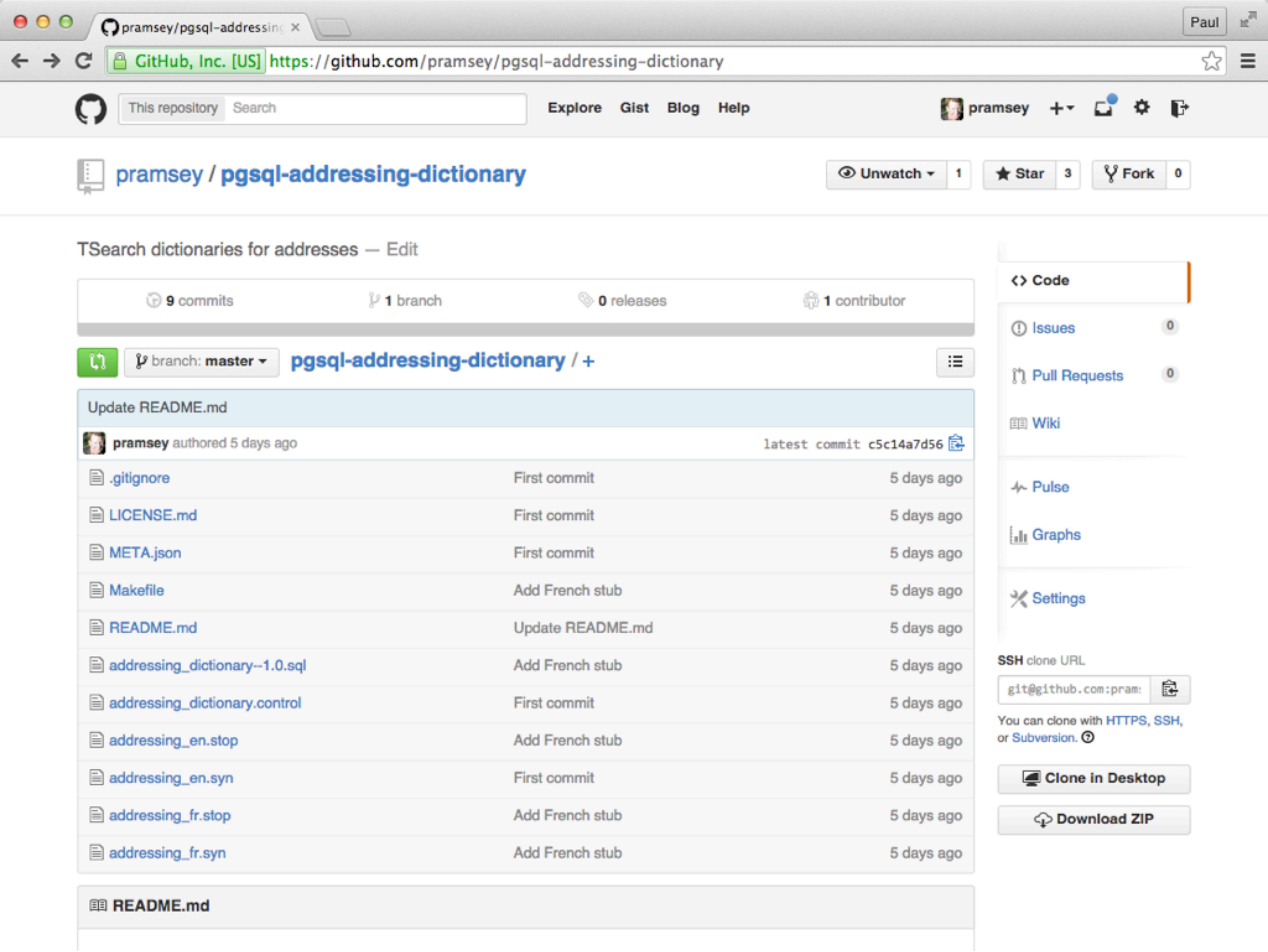
○ Aerial Imagery

○ Imagery with Streets

Overlays

☑ Tax Lots

What happens if we try to search for the names, **as they appear on the base map**?
"349 east main st" using the fully spelled out word "east", arg, no answers!
or JUST searching for "main street", but spelling street out in full
or searching for addresses on "south 2nd street", that appears on the map
no success!
what is going on here?
we have broken the street names into "words" and saved the "words" in the full-text engine, but the words aren't like english words, they have their own grammar and synonyms, so the search is failing.
CAN WE FIX IT?



pramsey / pgsql-addressing-dictionary

TSearch dictionaries for addresses — Edit

9 commits · 1 branch · 0 releases · 1 contributor

branch: master · pgsql-addressing-dictionary / +

Update README.md

pramsey authored 5 days ago · latest commit c5c14a7d56

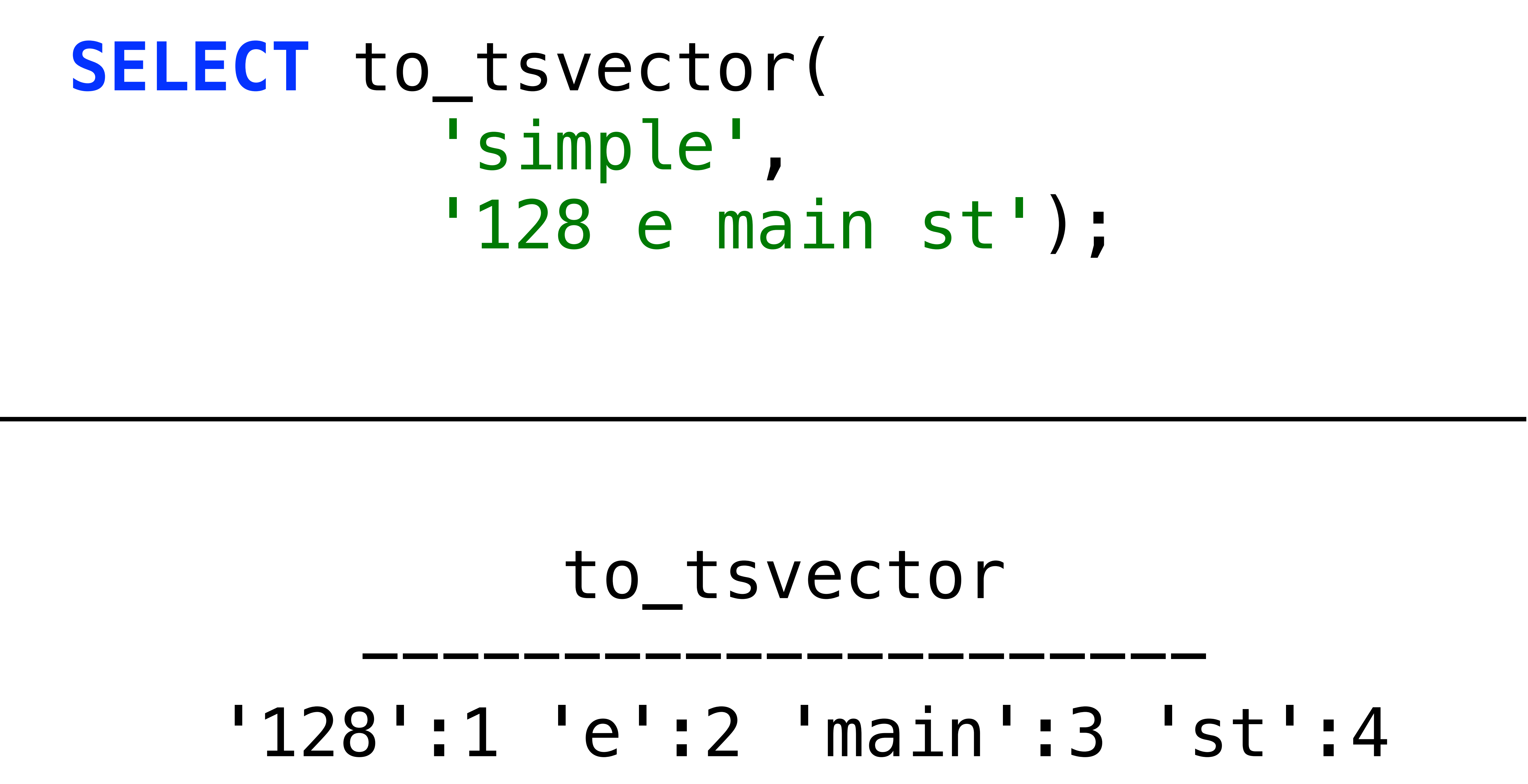
.gitignore	First commit	5 days ago
LICENSE.md	First commit	5 days ago
META.json	First commit	5 days ago
Makefile	Add French stub	5 days ago
README.md	Update README.md	5 days ago
addressing_dictionary--1.0.sql	Add French stub	5 days ago
addressing_dictionary.control	First commit	5 days ago
addressing_en.stop	Add French stub	5 days ago
addressing_en.syn	First commit	5 days ago
addressing_fr.stop	Add French stub	5 days ago
addressing_fr.syn	Add French stub	5 days ago

README.md

what if, instead of treating the words in the index as parts of language,
we treated them as parts of addresses?
so that the system would know that if you wrote “ST” you meant “STREET”
or if you wrote “N” you meant “NORTH”
then, searches using abbreviations would work,
or searches against **data** that was itself abbreviated would work

PostgreSQL tsearch allows you to create your own dictionaries
of synonyms, or words you want to ignore, or words you want
to replace with other words.

So I’ve created a custom dictionary, for street addresses.
The postgresql addressing dictionary.



```
SELECT to_tsvector(
          'simple',
          '128 e main st');
```

```
           to_tsvector
-------------------------------
'128':1 'e':2 'main':3 'st':4
```

for my basic example, I used the standard ‘simple’ dictionary set,
which doesn’t do **any** special processing on the words.
this is better than the ‘english’ dictionary,
which will drop things that aren’t english words (like ‘n’ or ‘st’) but it is still not that good,
since a search would have to use
exactly the same abbreviation style as the data to come up with a hit
so in this example
“128” is considered a word
“e” is considered a word
“st” it considered a word

```
CREATE EXTENSION
addressing_dictionary;

SELECT to_tsvector(
          'addressing_en',
          '128 e main st');
```

```
               to_tsvector
      ---------------------
 '128':1 'east':2 'main':3
        'street':4
```

But, when we parse the same thing using the addressing dictionary,
the custom address dictionary comes into play, and abbreviations are expanded out
‘e’ becomes ‘east’
‘st’ becomes ‘street’

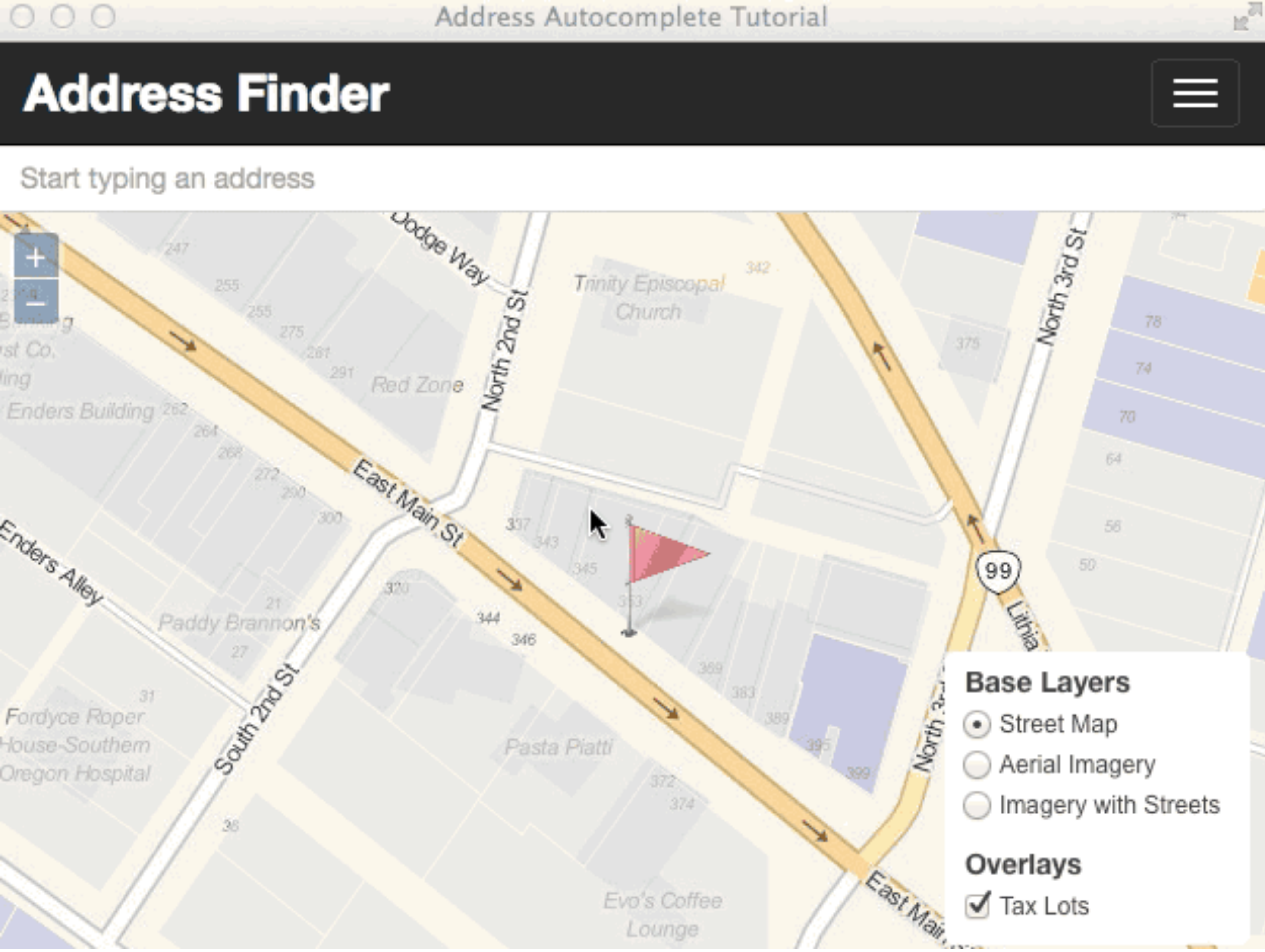
replace

'simple'

with

'addressing_en'

so, by altering the search application to just use the addressing dictionary instead, we can get much much better behaviour!



Base Layers

● Street Map

○ Aerial Imagery

○ Imagery with Streets

Overlays

☑ Tax Lots

“east main street” works
“south second street” works
things even work when the users mix up the “correct” addressing order and put the directions last
or even the house numbers last

Google Keywords

- postgresql fulltext search 9.4
- postgresql fulltext dictionary 9.4
- pramsey github

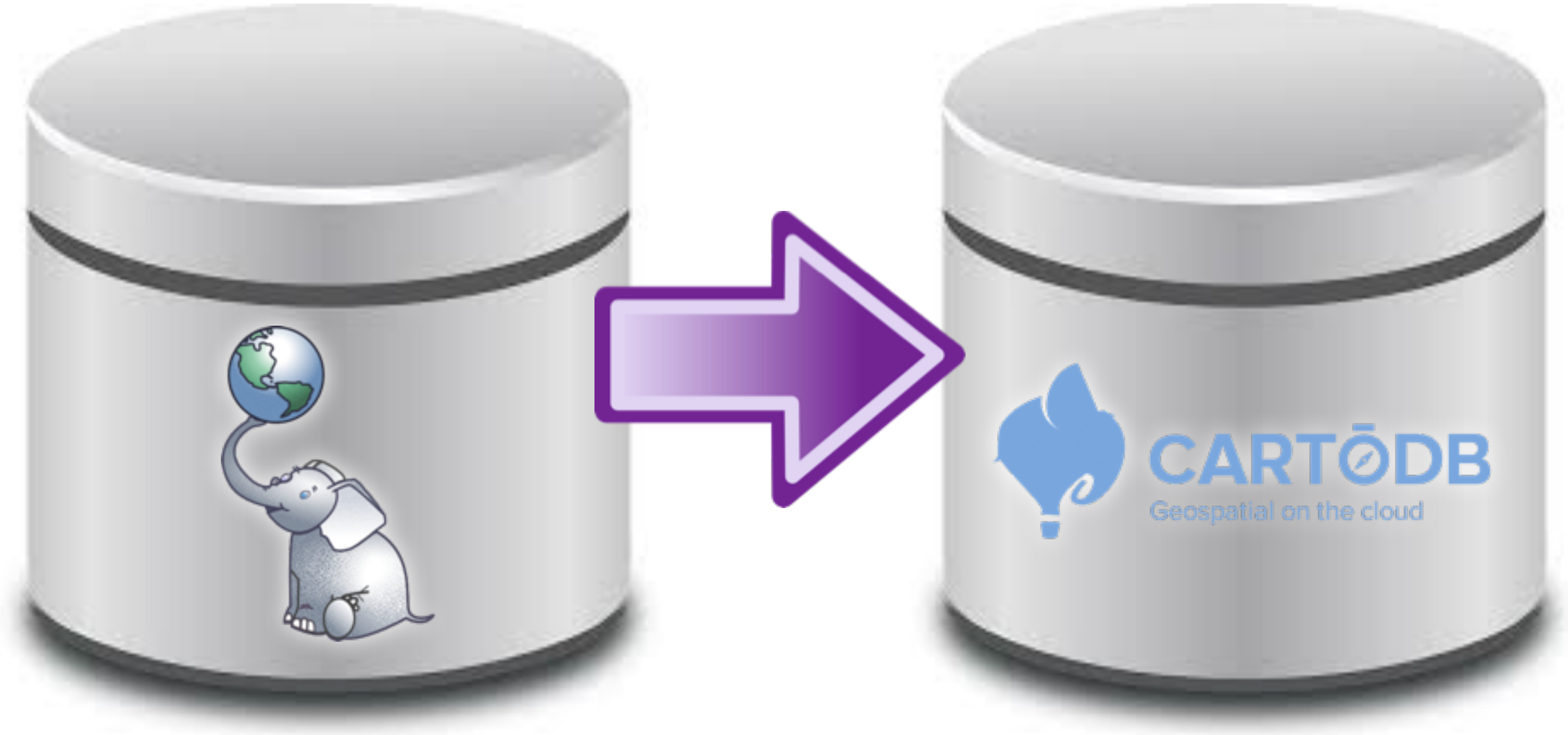
so, rather than give a bunch of URLs, here’s the modern version of the AOL keywords for this section,
“postgresql” “fulltext” “9.4” for the latest, but full-text has been part of PostgreSQL since 9.0
and “pramsey github” to find the addressing dictionaries to add to your database

2nd movement

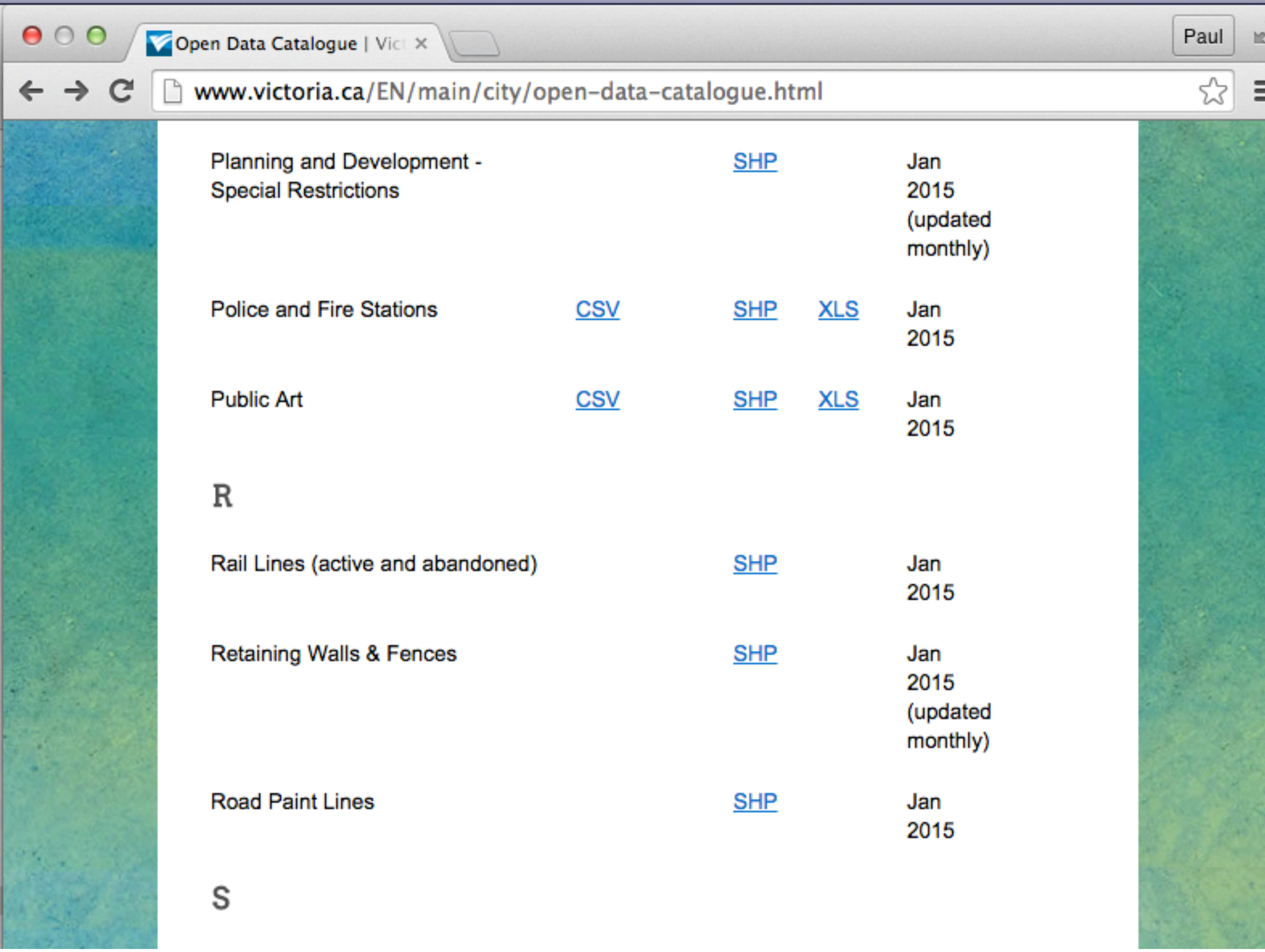
federation

so, a brief pause while the orchestra flips over the sheet music,
and on the the second movement: federated systems

data synchronization
"pushups"



first, "upwards" federation,
pushing data up from a local database into a "cloud" storage system,
and in deference to my employer,
and because it's so easy to sync to a system where there's no impedance mismatch
(copying from postgis to postgis is pretty easy)
I'll be showing how I federated a local postgis to CartoDB (a cloud postgis).


Open Data Catalogue | Vic
www.victoria.ca/EN/main/city/open-data-catalogue.html
Paul

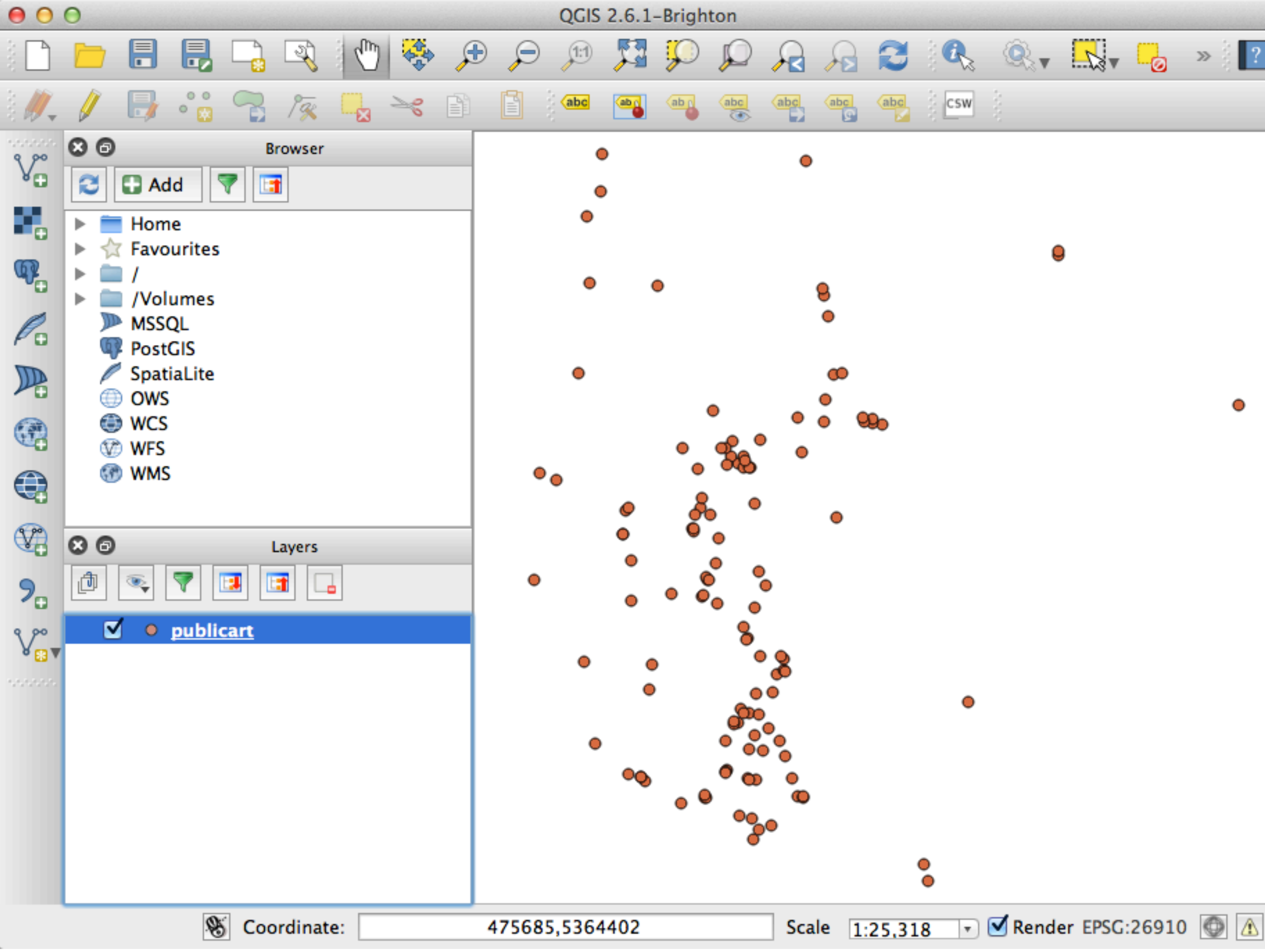
Planning and Development - Special Restrictions		SHP		Jan 2015 (updated monthly)
Police and Fire Stations	CSV	SHP	XLS	Jan 2015
Public Art	CSV	SHP	XLS	Jan 2015

R

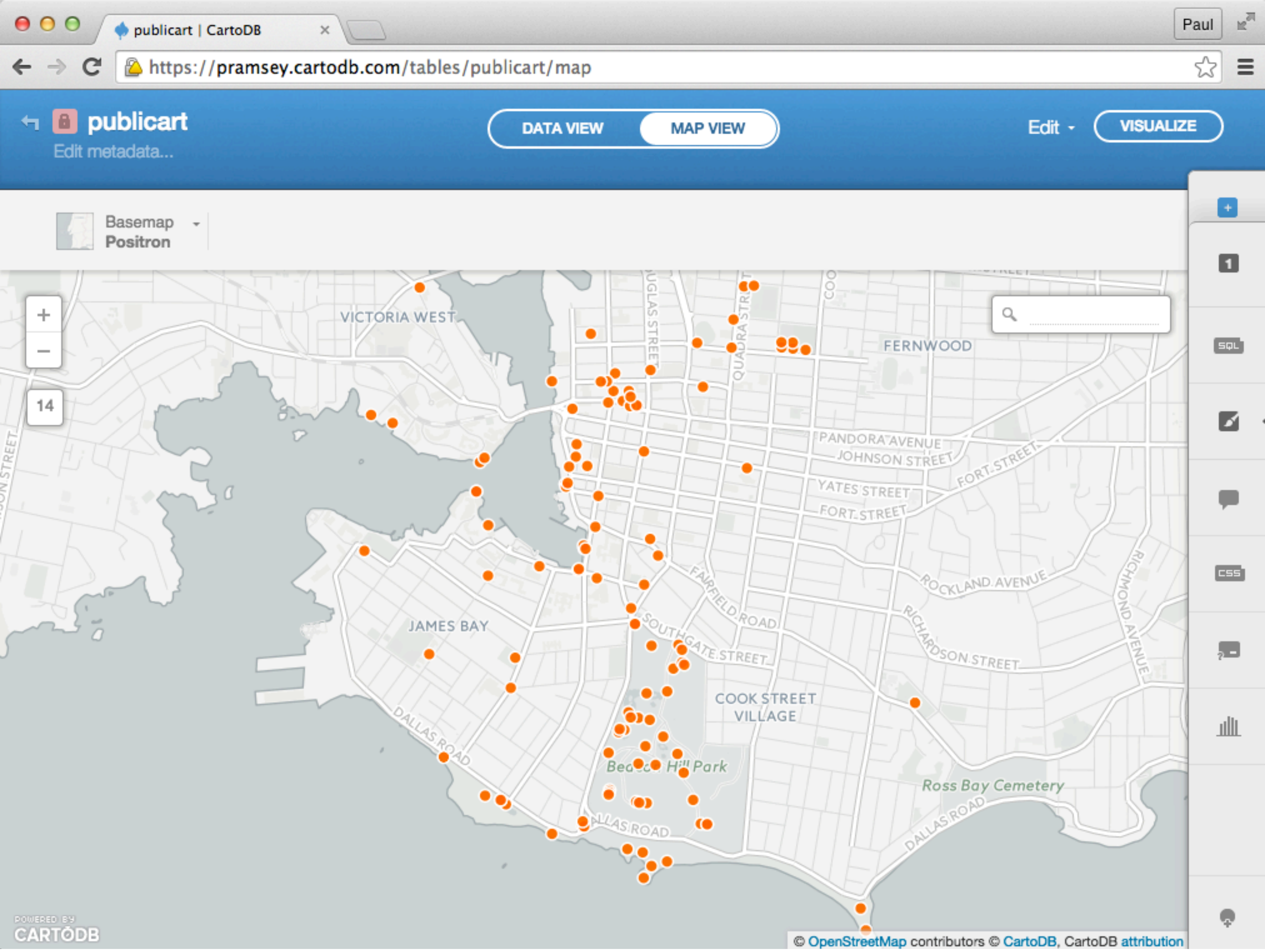
Rail Lines (active and abandoned)		SHP		Jan 2015
Retaining Walls & Fences		SHP		Jan 2015 (updated monthly)
Road Paint Lines		SHP		Jan 2015

S

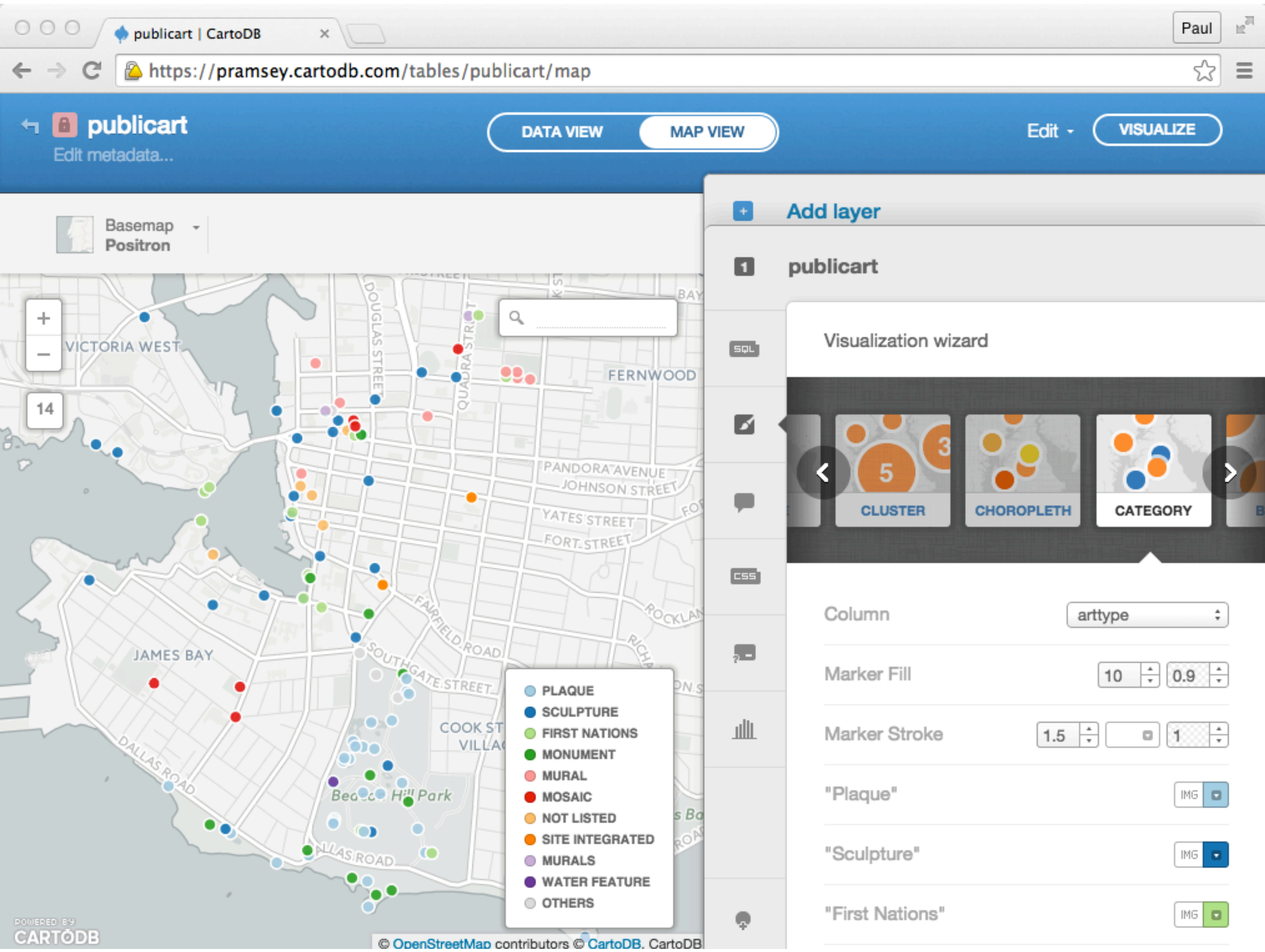
So, first I got some open data, from the City of Victoria, my home a shape file of Public Art in the city



And then I loaded it into my local PostGIS using shp2pgsql and viewed it in QGIS, there it is



And then I loaded the SAME data into CartoDB, and there it is, a little more comprehensible with the basemap underneath



● PLAQUE
● SCULPTURE
● FIRST NATIONS
● MONUMENT
● MURAL
● MOSAIC
● NOT LISTED
● SITE INTEGRATED
● MURALS
● WATER FEATURE
● OTHERS

1 publicart

Visualization wizard

Column: arttype

Marker Fill: 10 0.9

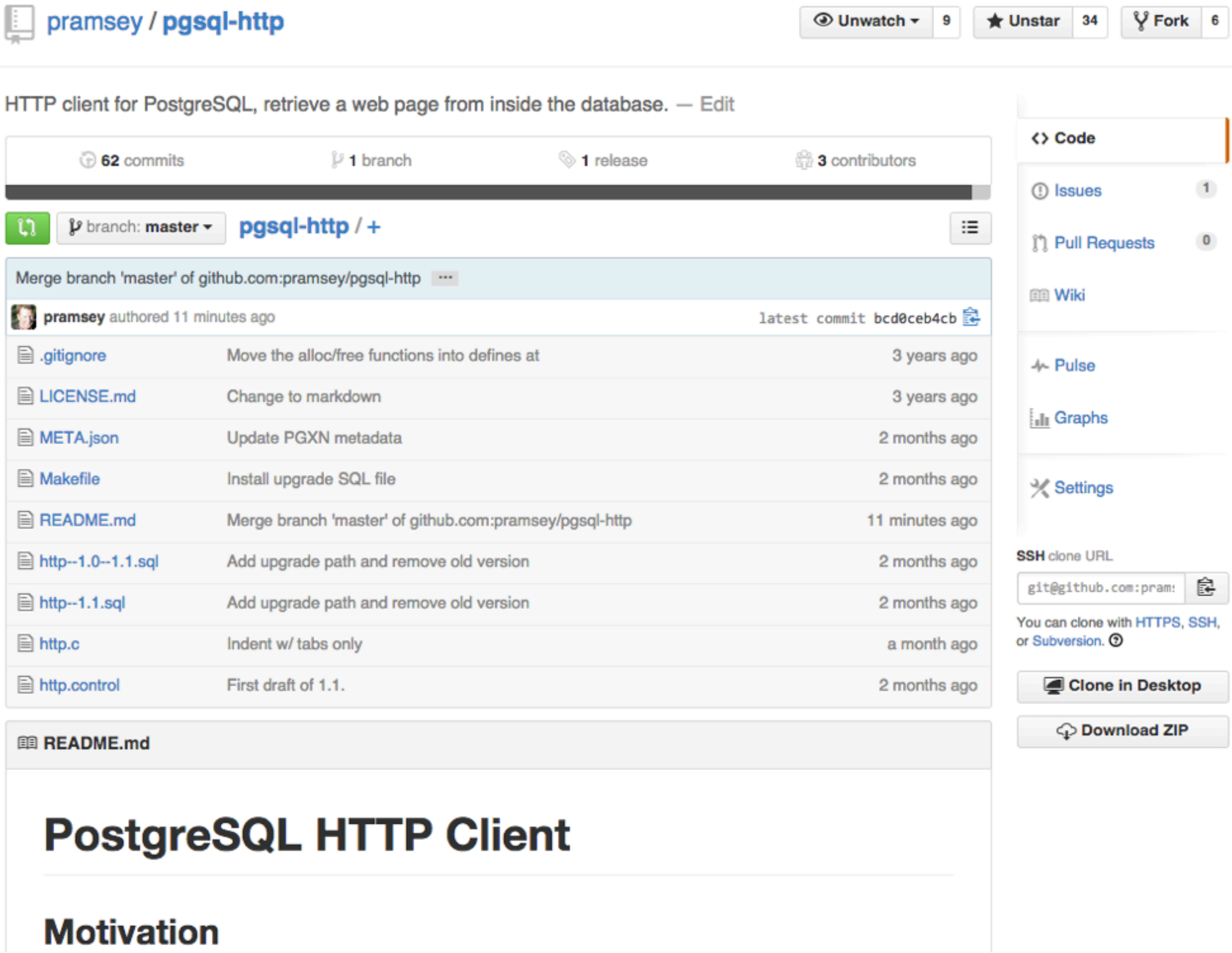
Marker Stroke: 1.5 1

"Plaque"

"Sculpture"

"First Nations"

And I can use the cartodb visualization tools to make it pretty, in this case with a categorical style
But, how do I connect the two systems?
How do I get changes in the local PostGIS to propogate
to the cloud CartoDB?
Well, CartoDB is a “web” service, so we need a “web” transport to push the changes over,
and as it happens, I wrote one of those!



HTTP client for PostgreSQL, retrieve a web page from inside the database. — Edit

Merge branch 'master' of github.com:pramsey/pgsql-http

pramsey authored 11 minutes ago — latest commit bcd0ceb4cb

.gitignore	Move the alloc/free functions into defines at	3 years ago
LICENSE.md	Change to markdown	3 years ago
META.json	Update PGXN metadata	2 months ago
Makefile	Install upgrade SQL file	2 months ago
README.md	Merge branch 'master' of github.com:pramsey/pgsql-http	11 minutes ago
http--1.0--1.1.sql	Add upgrade path and remove old version	2 months ago
http--1.1.sql	Add upgrade path and remove old version	2 months ago
http.c	Indent w/ tabs only	a month ago
http.control	First draft of 1.1.	2 months ago

SSH clone URL

git@github.com:pram:

You can clone with HTTPS, SSH, or Subversion.

README.md

PostgreSQL HTTP Client

Motivation

The HTTP extension for PostgreSQL. There’s nothing “spatial” about this extension, it just allows you to make HTTP calls using PostgreSQL functions.

```
SELECT content FROM http_get('http://localhost');

                 content
-------------------------------------------
 <html><body><h1>It works!</h1></body></html>
(1 row)
```

```
SELECT status,
  content_type,
  content
FROM http_get('http://localhost');

 status | content_type |                  content
--------+--------------+------------------------------
    200 | text/html    | <html><body><h1>It works!</h1></b
(1 row)
```

So you can run an “http_get()” function, and get back the results from a web service.
Not just the content, but also mime type, status codes, headers and so on.
And not just GET, either, but also POST, PUT and DELETE, so you can interact with any HTTP web service.
And here’s the thing:
CartoDB has a web service called... the “SQL API”. You can imagine what that is...

CARTODB
Geospatial on the cloud

```
http://pramsey.cartodb.com/api/v2/sql?
  format=json&
  api_key='<apikey>'&
  q='UPDATE+mytble+SET+mycol=myval+WHERE+something'
```

The SQL API is actually diabolically simple, you call an HTTP end point,
you tell it what format you want your return to be in (json or geojson)

If you’re altering the data, or the data is private, you provide an API key to prove who you are, and then you just provide the SQL you want executed!

It’s so diabolical, I actually described it, a couple years before it was invented, as

CARTODB
Geospatial on the cloud

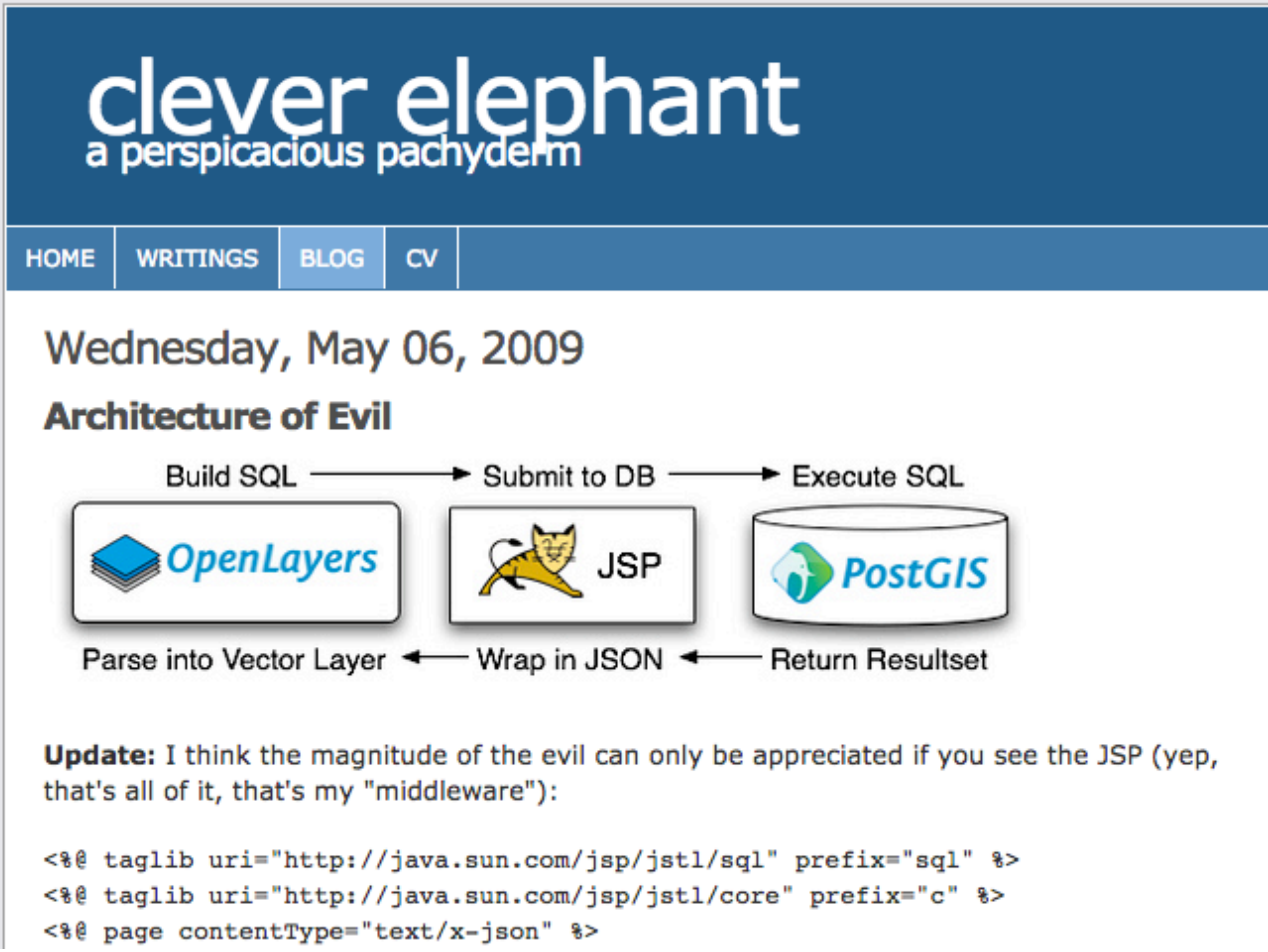
CartoDB
SQL API

```
http://pramsey.cartodb.com/api/v2/sql?
  format=json&
  api_key='<apikey>'&
  q='UPDATE+mytble+SET+mycol=myval+WHERE+something'
```

The SQL API is actually diabolically simple, you call an HTTP end point,
you tell it what format you want your return to be in (json or geojson)

If you're altering the data, or the data is private, you provide an API key to prove who you are, and then you just provide the SQL you want executed!

It's so diabolical, I actually described it, a couple years before it was invented, as



Wednesday, May 06, 2009

Architecture of Evil

Update: I think the magnitude of the evil can only be appreciated if you see the JSP (yep, that's all of it, that's my "middleware"):

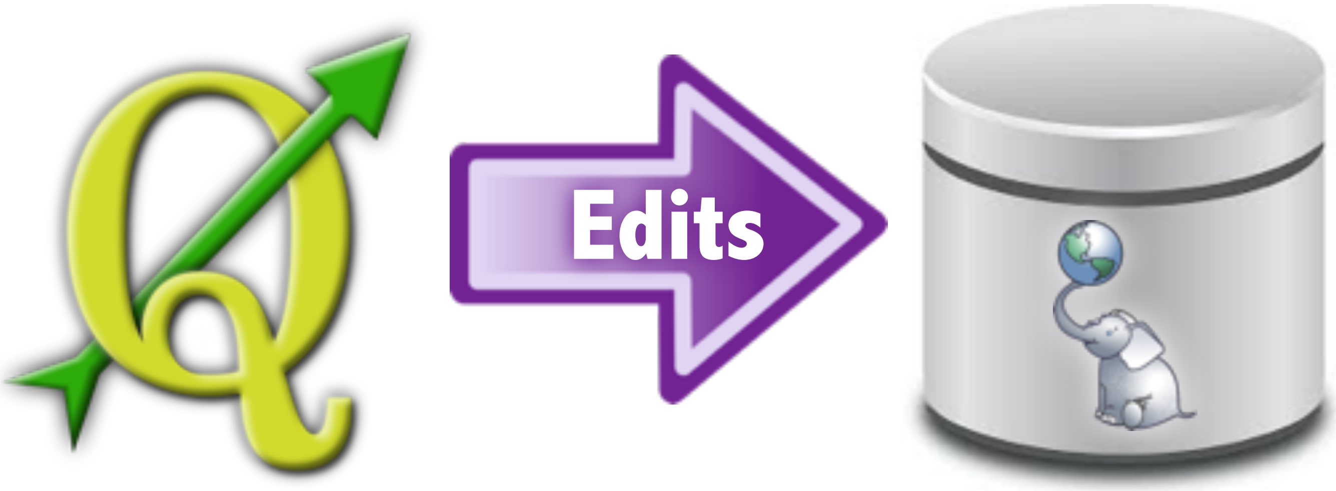
```
<%@ taglib uri="http://java.sun.com/jsp/jstl/sql" prefix="sql" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ page contentType="text/x-json" %>
```

The “architecture of evil”, since with an unprotected SQL passthrough there’s so much evil the outside world could work on your database,

Of course the CartoDB API is protected against SQL injection and users are isolated in their own databases, so it’s not really the same thing as I described in 2009, which was incredibly lightweight and passed the SQL into the database completely un-inspected.

But the simplicity of the approach allows for incredibly flexibility in building applications, since there’s no need at the HTTP interface level to re-invent things to proxy for SQL. (Which is what the OGC WFS specification did)

So, for this example of federation, I used QGIS as an editor and <X> directly edited a local PostGIS database.
Each database UPDATE in turn <X> triggered an http_post() call, using the HTTP extension, which passed the <X> UDPATE to the CartoDB SQL API.
This in turn was <X> applied to the CartoDB database which made it visible to me in <X> Chrome looking at the CartoDB rendering.
Diabolical. Pure evil.

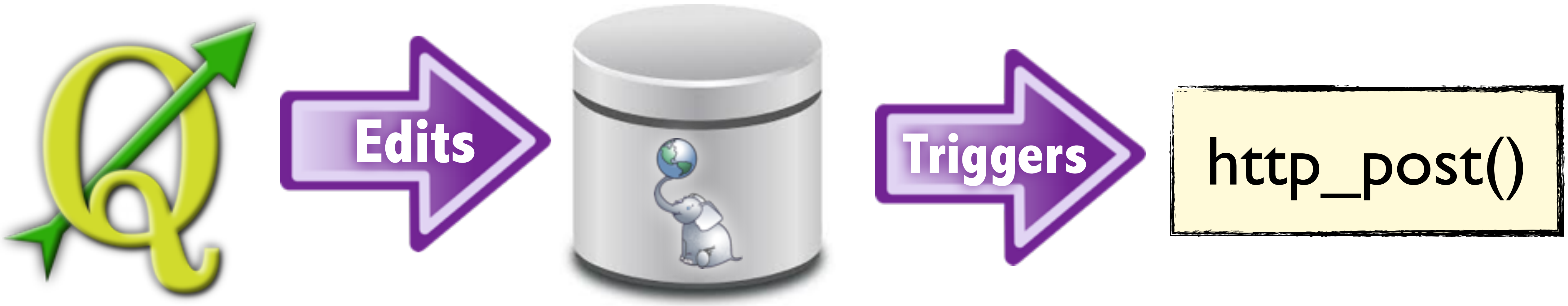


So, for this example of federation, I used QGIS as an editor and <X> directly edited a local PostGIS database.

Each database UPDATE in turn <X> triggered an http_post() call, using the HTTP extension, which passed the <X> UDPATE to the CartoDB SQL API.

This in turn was <X> applied to the CartoDB database which made it visible to me in <X> Chrome looking at the CartoDB rendering.

Diabolical. Pure evil.

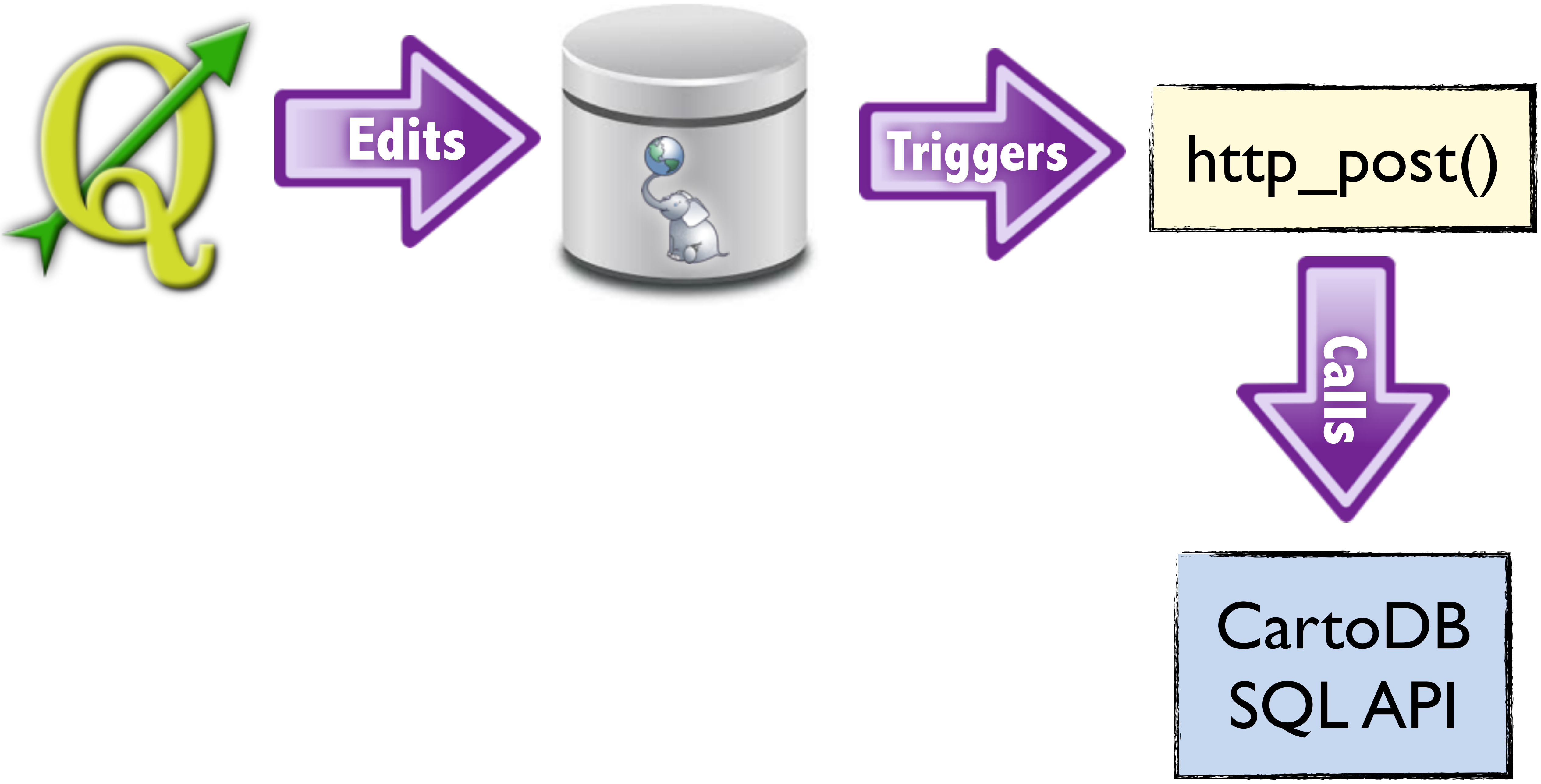


Edits

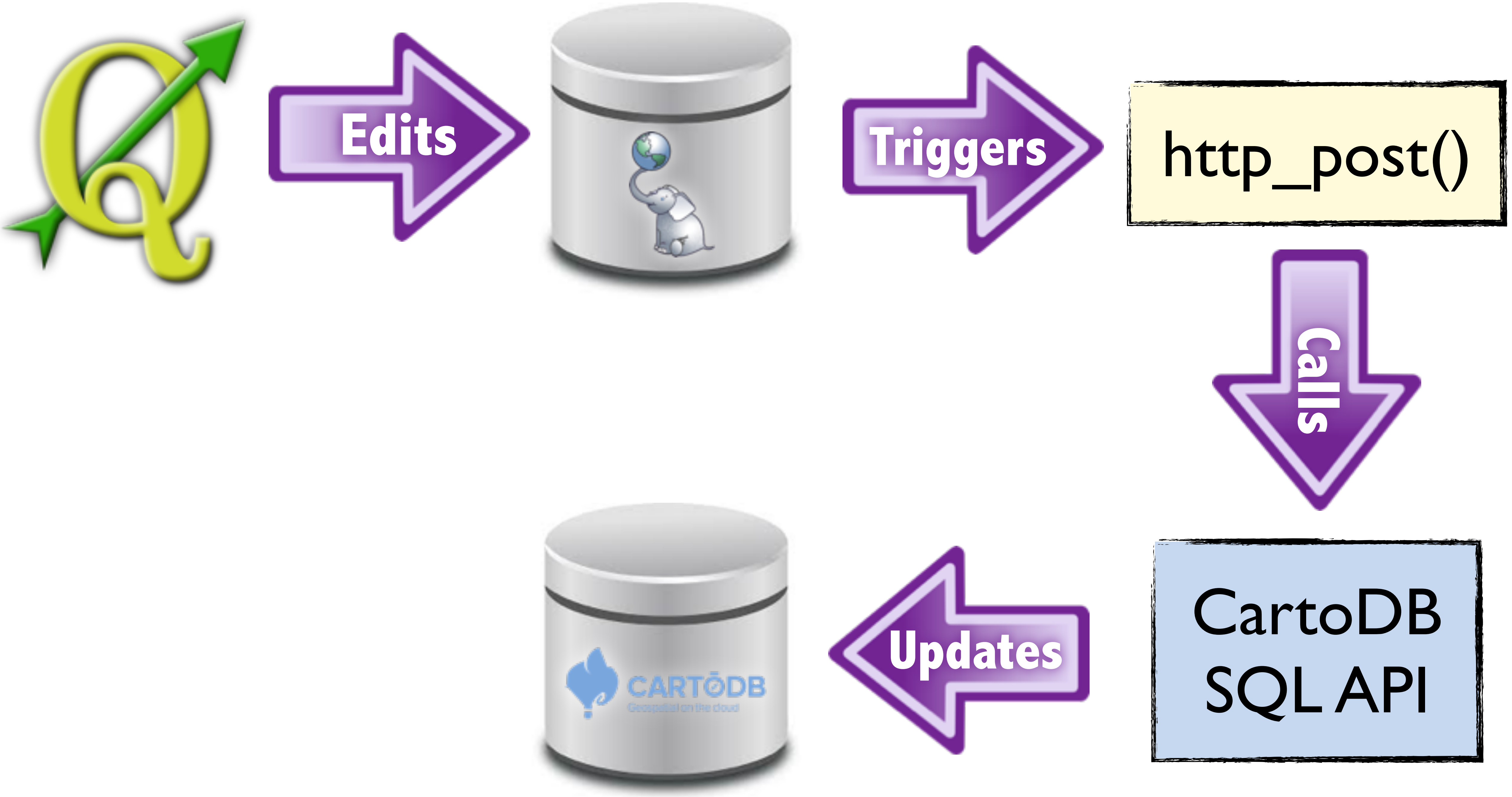
Triggers

http_post()

So, for this example of federation, I used QGIS as an editor and <X> directly edited a local PostGIS database.
Each database UPDATE in turn <X> triggered an http_post() call, using the HTTP extension, which passed the <X> UDPATE to the CartoDB SQL API.
This in turn was <X> applied to the CartoDB database which made it visible to me in <X> Chrome looking at the CartoDB rendering.
Diabolical. Pure evil.



So, for this example of federation, I used QGIS as an editor and <X> directly edited a local PostGIS database.
Each database UPDATE in turn <X> triggered an http_post() call, using the HTTP extension, which passed the <X> UDPATE to the CartoDB SQL API.
This in turn was <X> applied to the CartoDB database which made it visible to me in <X> Chrome looking at the CartoDB rendering.
Diabolical. Pure evil.

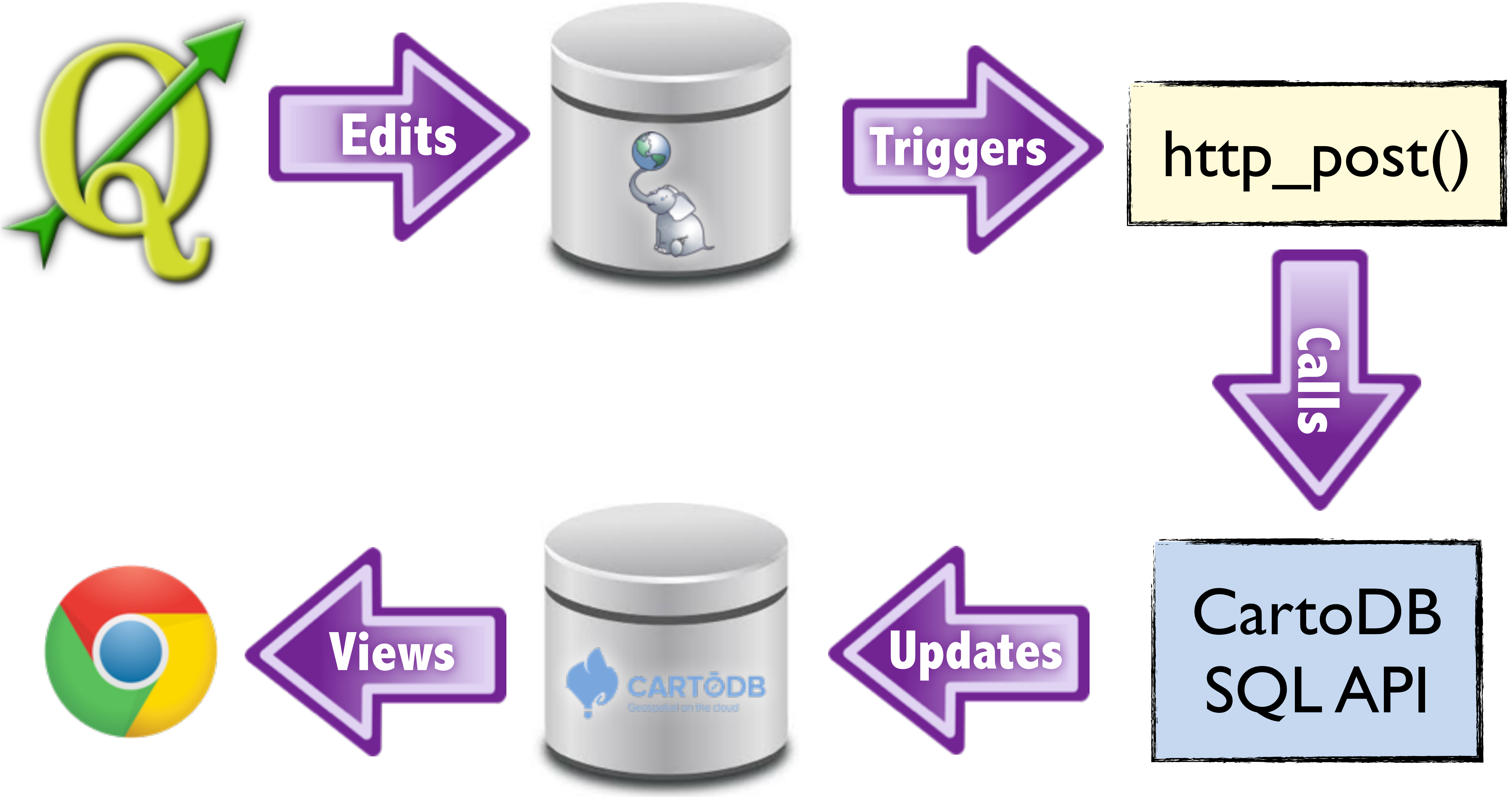


So, for this example of federation, I used QGIS as an editor and <X> directly edited a local PostGIS database.

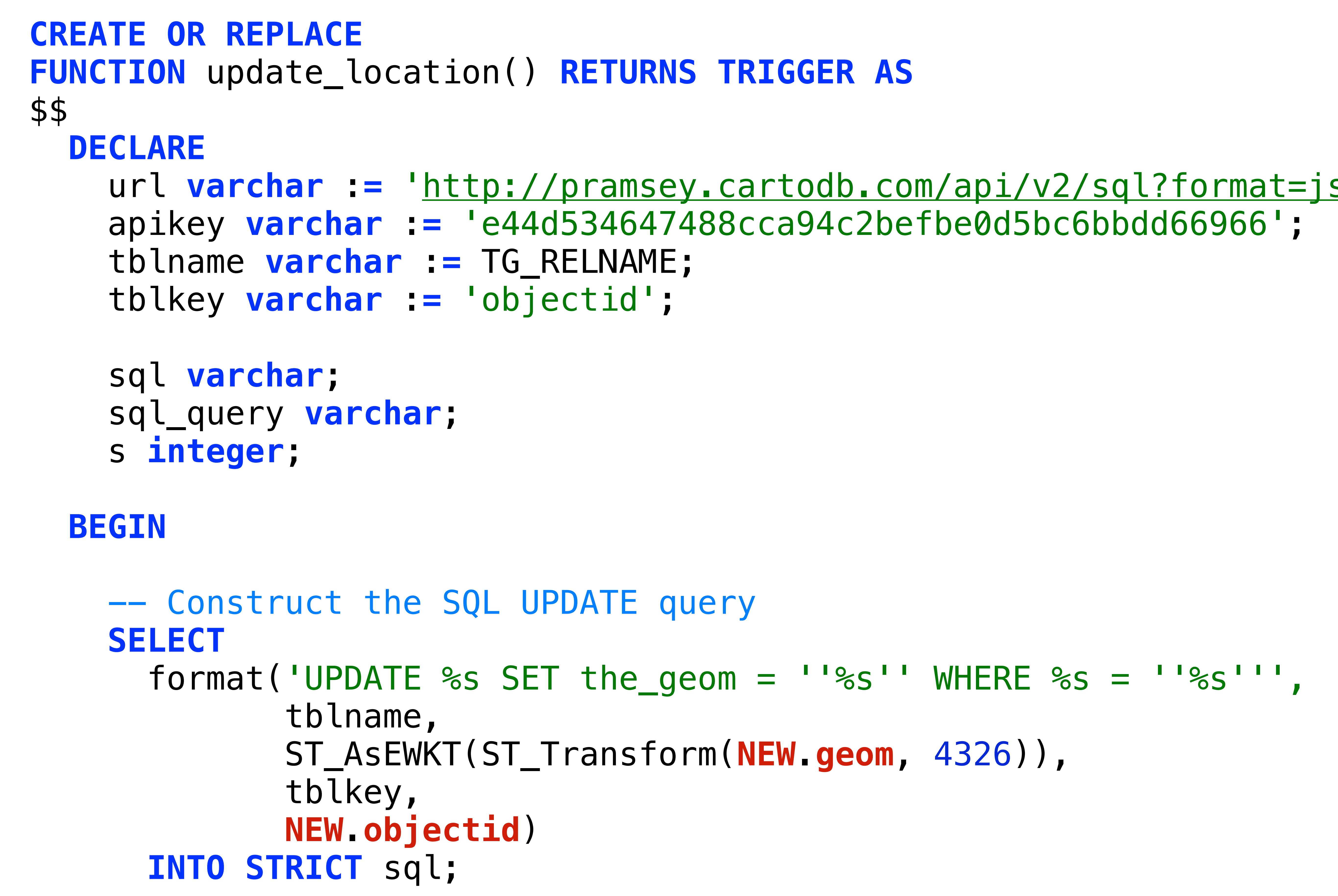
Each database UPDATE in turn <X> triggered an http_post() call, using the HTTP extension, which passed the <X> UDPATE to the CartoDB SQL API.

This in turn was <X> applied to the CartoDB database which made it visible to me in <X> Chrome looking at the CartoDB rendering.

Diabolical. Pure evil.



So, for this example of federation, I used QGIS as an editor and <X> directly edited a local PostGIS database.
Each database UPDATE in turn <X> triggered an http_post() call, using the HTTP extension, which passed the <X> UDPATE to the CartoDB SQL API.
This in turn was <X> applied to the CartoDB database which made it visible to me in <X> Chrome looking at the CartoDB rendering.
Diabolical. Pure evil.



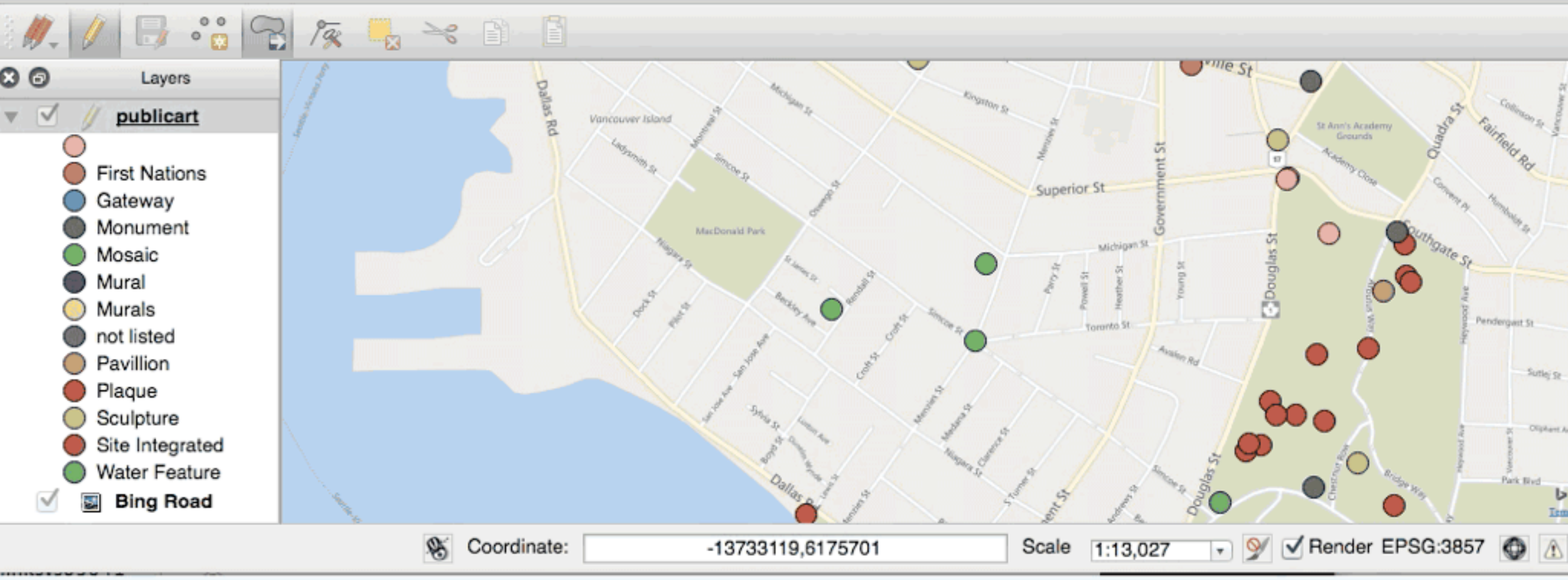
```
CREATE OR REPLACE
FUNCTION update_location() RETURNS TRIGGER AS
$$
  DECLARE
    url varchar := 'http://pramsey.cartodb.com/api/v2/sql?format=js
    apikey varchar := 'e44d534647488cca94c2befbe0d5bc6bbdd66966';
    tblname varchar := TG_RELNAME;
    tblkey varchar := 'objectid';

    sql varchar;
    sql_query varchar;
    s integer;

  BEGIN

    -- Construct the SQL UPDATE query
    SELECT
      format('UPDATE %s SET the_geom = ''%s'' WHERE %s = ''%s''',
             tblname,
             ST_AsEWKT(ST_Transform(NEW.geom, 4326)),
             tblkey,
             NEW.objectid)
      INTO STRICT sql;
```

so, here’s the local database trigger that updates CartoDB
- it’s only tied to update events, but if we were doing full CRUD we could make an insert and a delete version too
- to WRITE to CartoDB we need to authenticate, so we provide a key
- the SQL to update the table is simple, since we’re only updating the location field (a more complete version might update all fields)

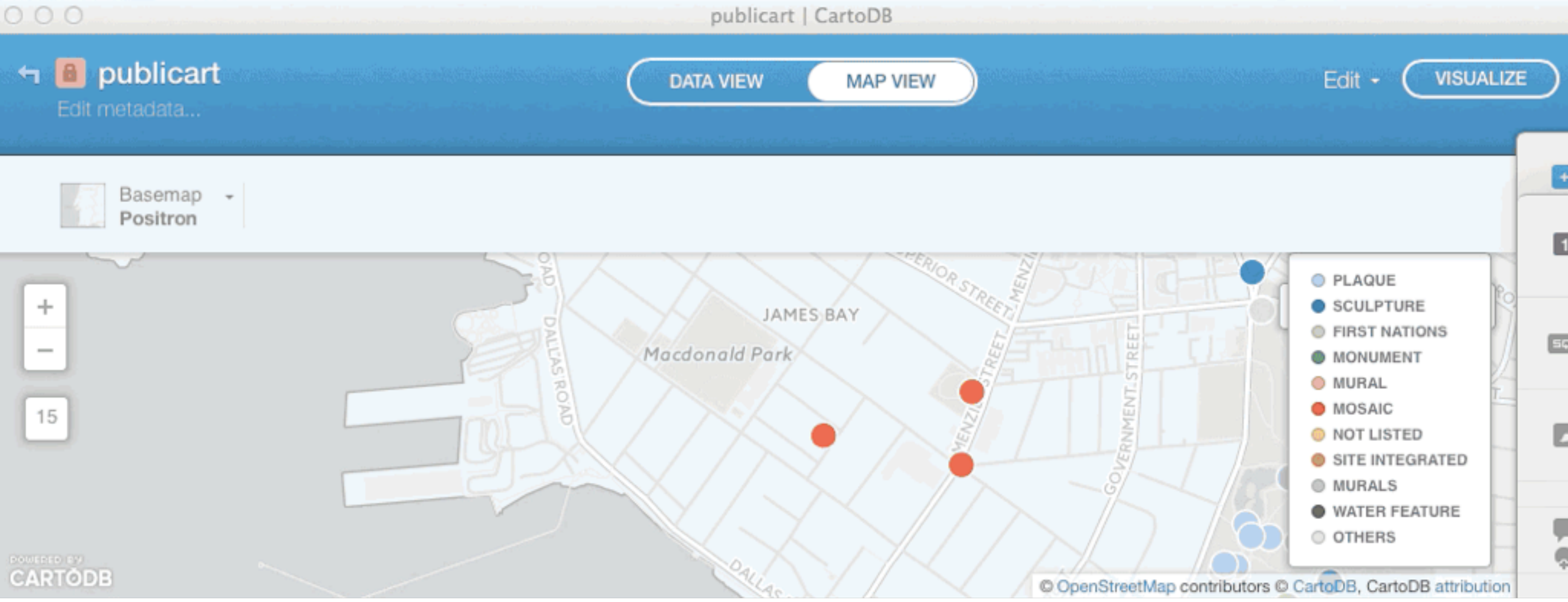
```
    -- URL encode the query so we can send it over the wire
    SELECT 'q=' || urlencode(sql)
      INTO STRICT sql_query;

    -- POST the query to CartoDB
    SELECT status
      INTO STRICT s
      FROM http_post(url || apikey,
                     sql_query,
                     'application/x-www-form-urlencoded');

    -- Don't commit unless the update succeeds
    IF s != 200 THEN
      RAISE EXCEPTION
        'HTTP POST to % using % failed (%)', url, sql, s;
    END IF;

    RETURN NEW;

  END;
$$
LANGUAGE 'plpgsql';
```

- we have to URL encode the SQL to pass it through the HTTP form
- then run the HTTP POST up to CartoDB
- check the return code for a good response
- and return the NEW tuple for writing to the local database!

that’s it!



Coordinate: -13733119,6175701 Scale 1:13,027 ☑ Render EPSG:3857



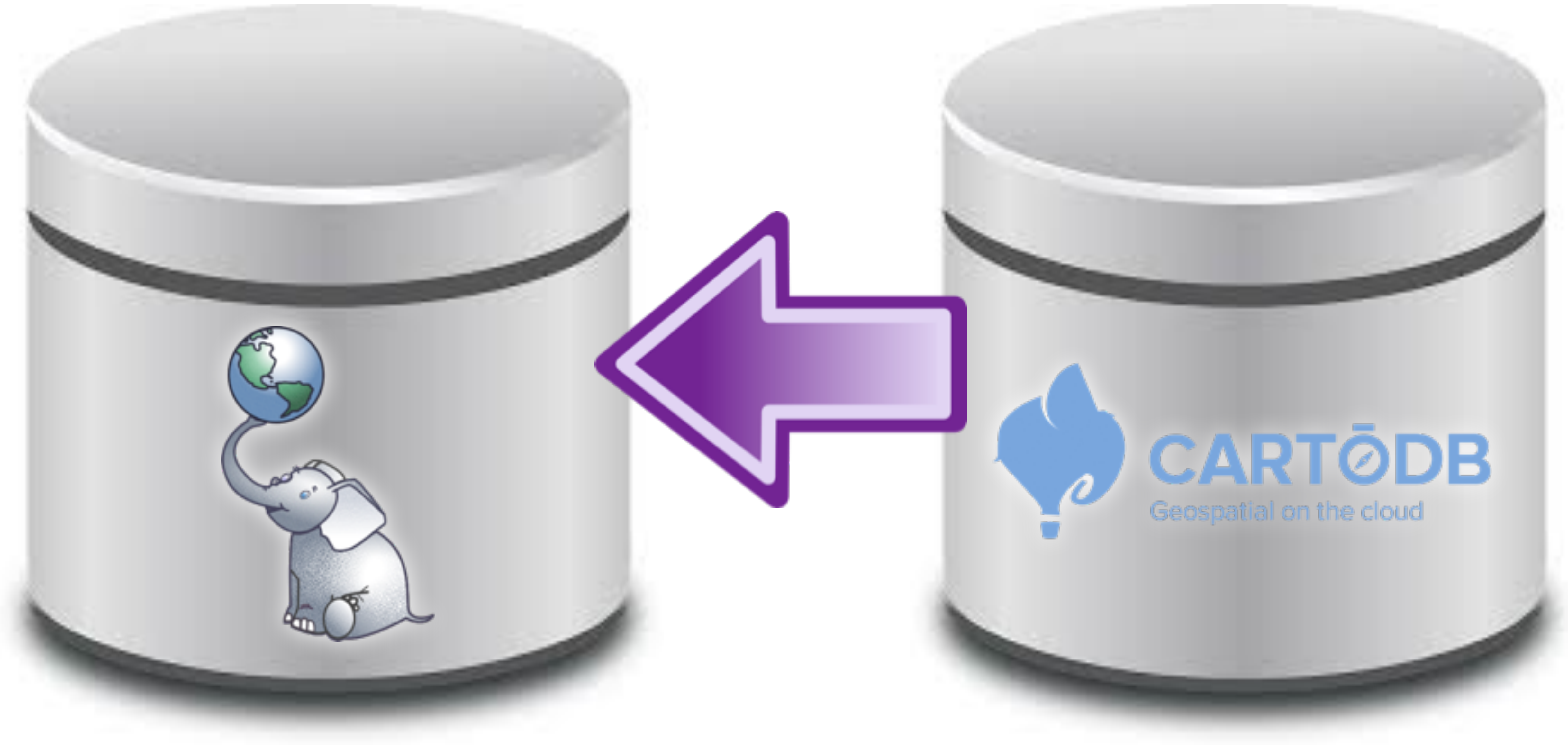
and here it is in action
I positioned my QGIS window on the top for editing, and my CartoDB map on the bottom for seeing the results
I edit a point and move it,
then refresh the CartoDB window, and see the change has been sent up!

Martin Jensen (@kukkaide)

```
COPY
(SELECT ST_X(geom) AS lon, ST_Y(geom) AS lat, name, title
 FROM data.locationtable
  WHERE title LIKE 'B%')
 TO PROGRAM '
 cat <&0 > /tmp/data.csv;
 curl -F "file=@/tmp/data.csv;filename=cartodbtablename.csv"
   "https://{CARTODB_USER}.cartodb.com/api/v1/imports/?api_key={APIKEY}"'
 DELIMITER ','
 CSV HEADER;
```

The trigger method is a nice incremental solution, but if you want something even simpler operating in batch, this solution from Martin Jensen is even more DIABOLICAL,
it dumps a table directly into Curl (this example uses CSV, so it’s only good for points) and slams the table right onto the CartoDB Import API

data synchronization
"pushdowns"

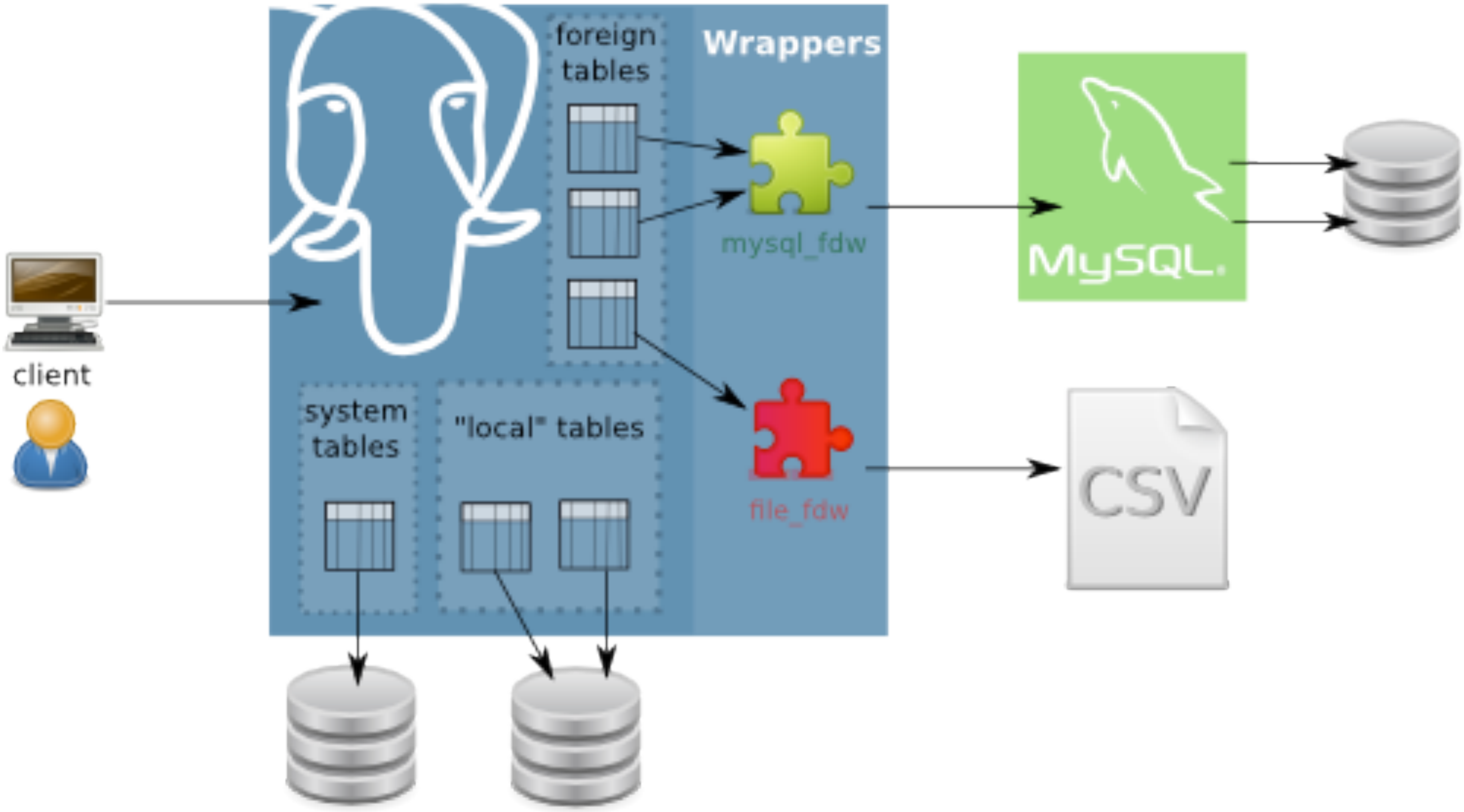


So, that was an example of a "push UP", moving data from a local PostGIS to a remote HTTP host. How about a push DOWN? Pushing data DOWN from the cloud into a local PostGIS database? Can we do the reverse?

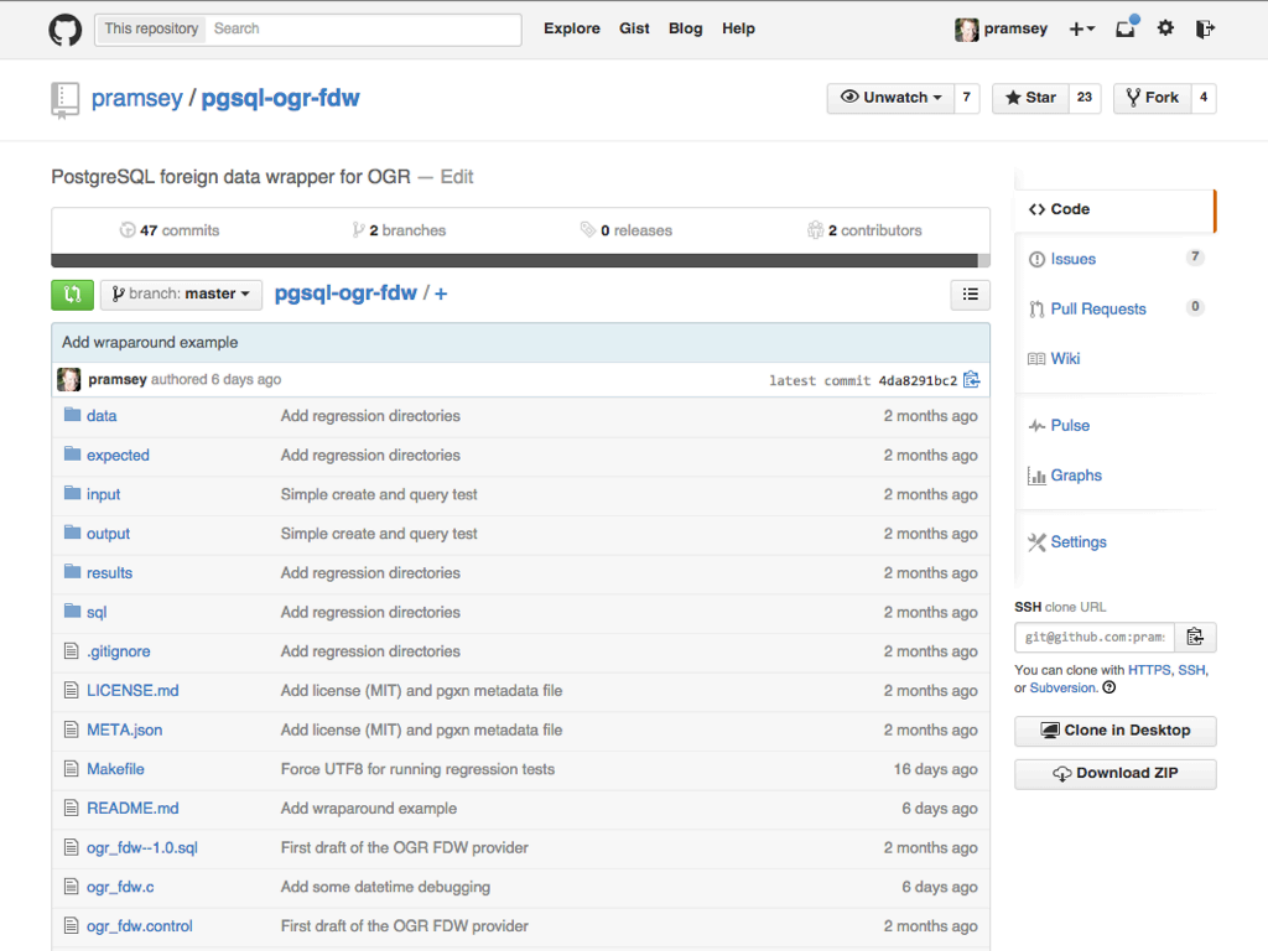
SQL/MED, or *Management of External Data*, extension to the SQL standard is defined by ISO/ IEC 9075-9:2003

Sure, we can! Using a fancy SQL standard, called MED, “management of external data”, which is exposed in PostgreSQL as a real world piece of functionality called

FDW



"foreign data wrappers" or "FDW"
foreign data wrappers expose what looks to a client just like a table in the database. You access the data by running SELECT queries on it, up change it by running INSERT/UPDATE and DELETE commands on it. But behind the scenes, a foreign data wrapper table can be anything at all.
It can be a table on a remote database, and not necessarily even a remote PostgreSQL database. There are wrappers for Oracle and MySQL and others.
Or it could be a non-database data source, like flat file.
Or it could be a non-tabular data source, like a twitter query.
There are FDW implementations for all these things.



pramsey / pgsql-ogr-fdw

PostgreSQL foreign data wrapper for OGR — Edit

47 commits · 2 branches · 0 releases · 2 contributors

branch: master · pgsql-ogr-fdw / +

Add wraparound example

pramsey authored 6 days ago · latest commit 4da8291bc2

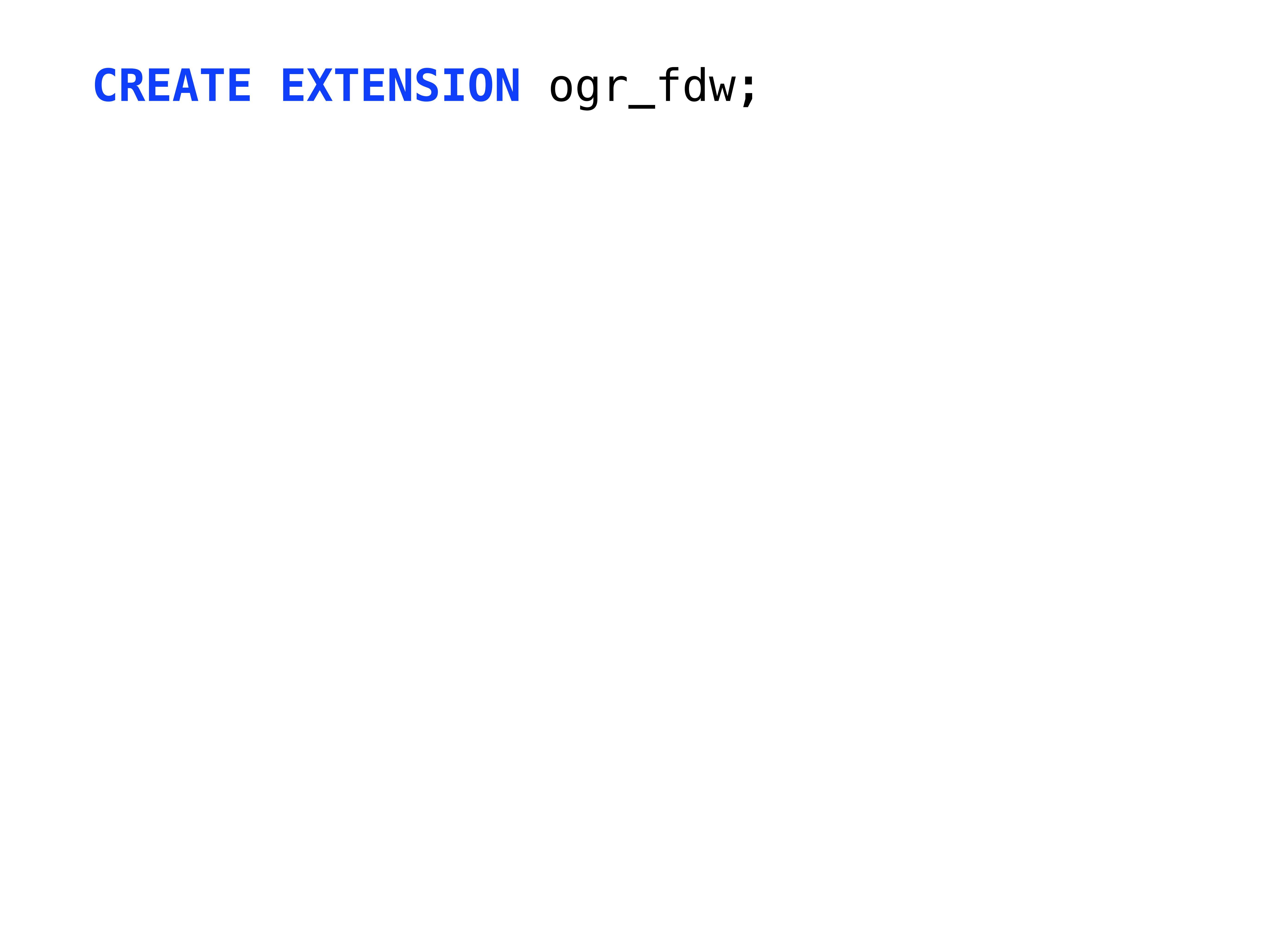
data	Add regression directories	2 months ago
expected	Add regression directories	2 months ago
input	Simple create and query test	2 months ago
output	Simple create and query test	2 months ago
results	Add regression directories	2 months ago
sql	Add regression directories	2 months ago
.gitignore	Add regression directories	2 months ago
LICENSE.md	Add license (MIT) and pgxn metadata file	2 months ago
META.json	Add license (MIT) and pgxn metadata file	2 months ago
Makefile	Force UTF8 for running regression tests	16 days ago
README.md	Add wraparound example	6 days ago
ogr_fdw--1.0.sql	First draft of the OGR FDW provider	2 months ago
ogr_fdw.c	Add some datetime debugging	6 days ago
ogr_fdw.control	First draft of the OGR FDW provider	2 months ago

SSH clone URL

git@github.com:pram:

You can clone with HTTPS, SSH, or Subversion.

However, the one I’m going to talk about today is an FDW wrapper for OGR, the spatial data abstraction library.
It’s a perfect fit for an FDW wrapper in many ways, since it exposes a very “tabular” kind of data, spatial layers.
And since it is a multi-format spatial library, by impementing an OGR FDW, we get access to many formats for the price of writing just one wrapper.

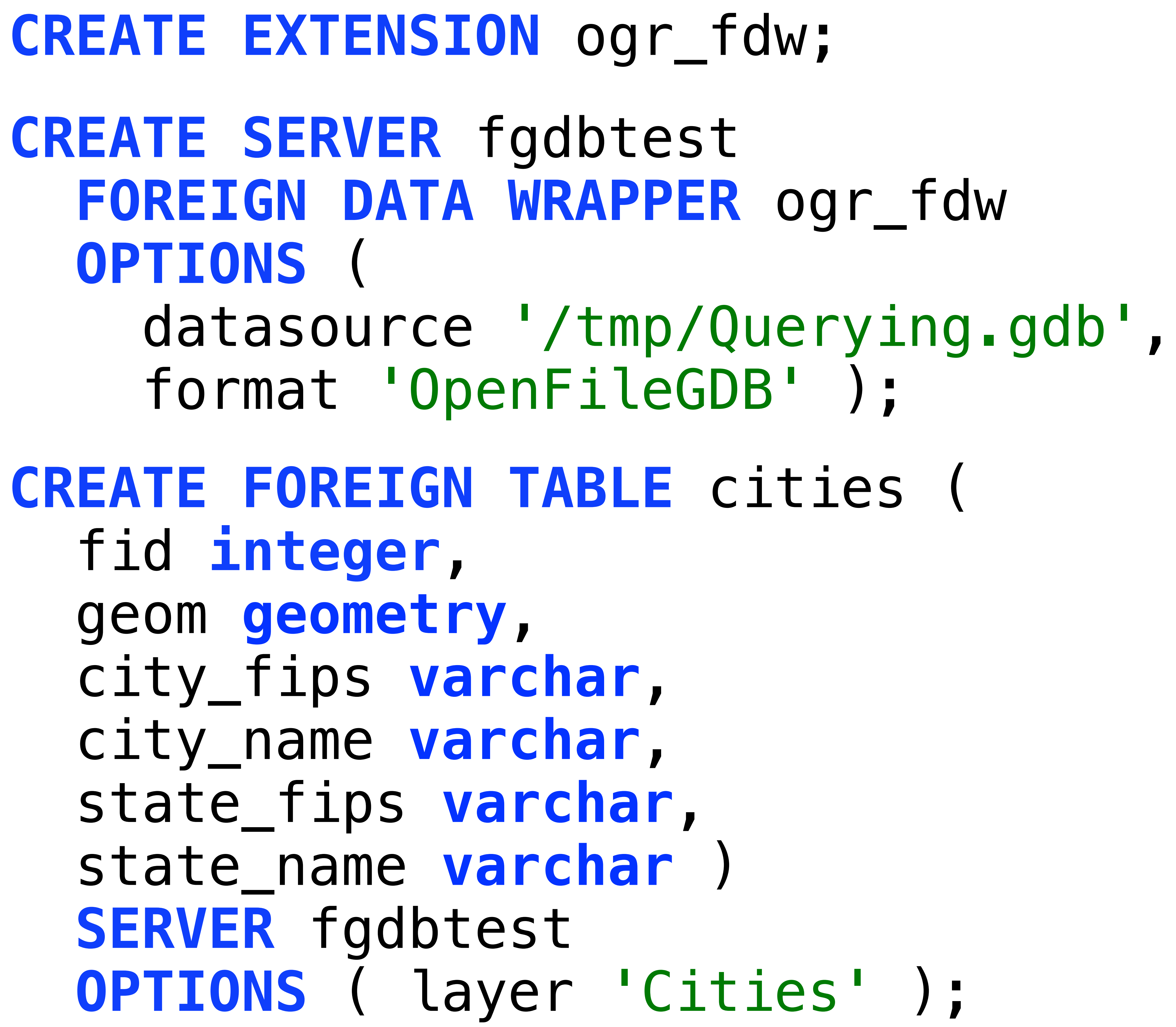


```
CREATE EXTENSION ogr_fdw;
```

Here’s what it looks like to expose a file geodatabase to PostgreSQL using the OGR FDW.
First you turn on the ogr fdw extension.
Then, you create a “server” that references the data source, in this case an FGDB file. You can see that the nomenclature really assumes you’ll be working against other database servers, but fortunately there is no real restriction.
Finally, you create a foreign table that in turn references the server. It defines what columns from the foreign server you want to expose in your local database.

```
CREATE EXTENSION ogr_fdw;

CREATE SERVER fgdbtest
  FOREIGN DATA WRAPPER ogr_fdw
  OPTIONS (
    datasource '/tmp/Querying.gdb',
    format 'OpenFileGDB' );
```

Here’s what it looks like to expose a file geodatabase to PostgreSQL using the OGR FDW.
First you turn on the ogr fdw extension.
Then, you create a “server” that references the data source, in this case an FGDB file. You can see that the nomenclature really assumes you’ll be working against other database servers, but fortunately there is no real restriction.
Finally, you create a foreign table that in turn references the server. It defines what columns from the foreign server you want to expose in your local database.

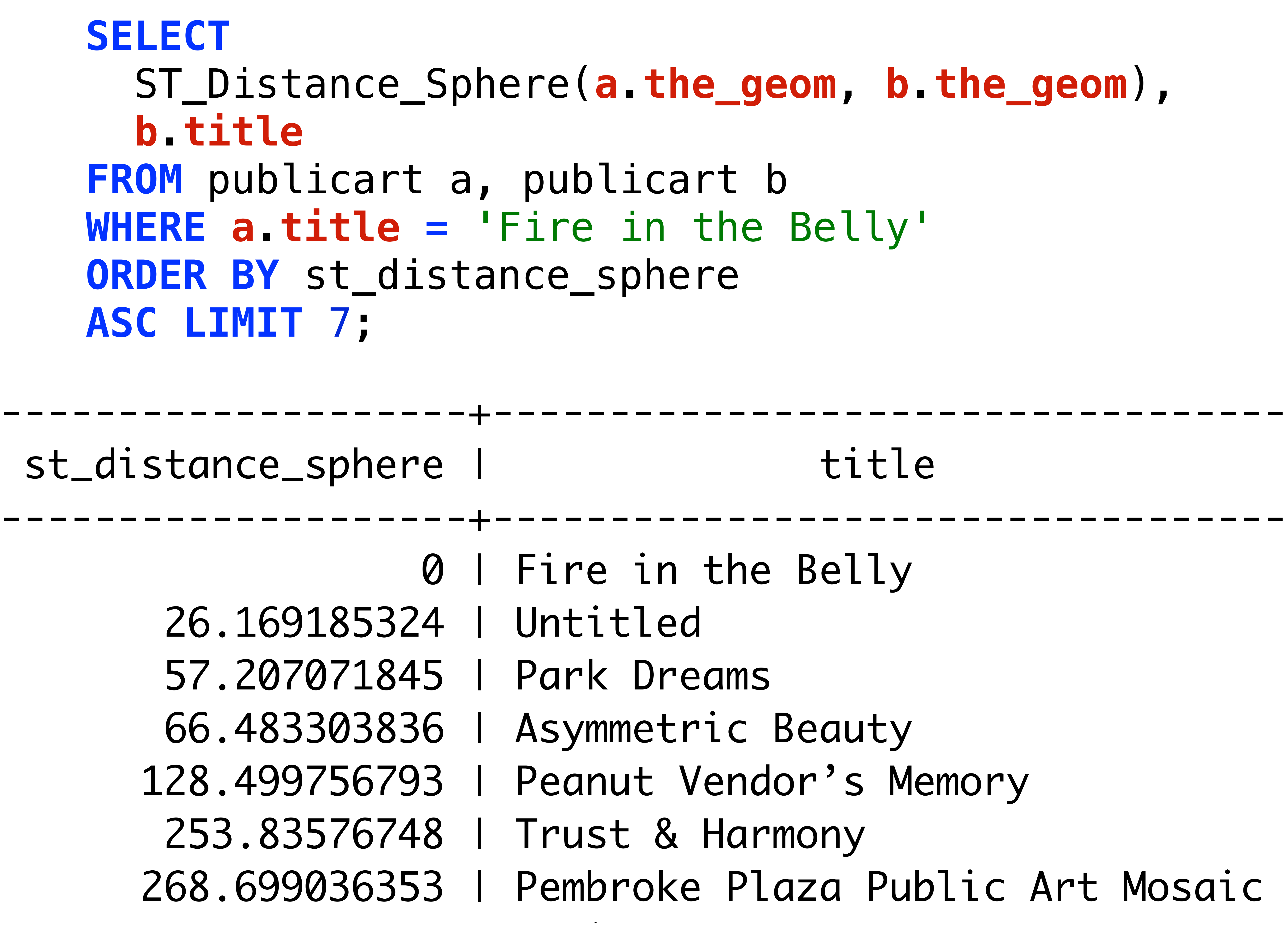


```
CREATE EXTENSION ogr_fdw;

CREATE SERVER fgdbtest
  FOREIGN DATA WRAPPER ogr_fdw
  OPTIONS (
    datasource '/tmp/Querying.gdb',
    format 'OpenFileGDB' );

CREATE FOREIGN TABLE cities (
  fid integer,
  geom geometry,
  city_fips varchar,
  city_name varchar,
  state_fips varchar,
  state_name varchar )
  SERVER fgdbtest
  OPTIONS ( layer 'Cities' );
```

Here’s what it looks like to expose a file geodatabase to PostgreSQL using the OGR FDW.
First you turn on the ogr fdw extension.
Then, you create a “server” that references the data source, in this case an FGDB file. You can see that the nomenclature really assumes you’ll be working against other database servers, but fortunately there is no real restriction.
Finally, you create a foreign table that in turn references the server. It defines what columns from the foreign server you want to expose in your local database.

```
CREATE SERVER cartodb
  FOREIGN DATA WRAPPER ogr_fdw
  OPTIONS (
    datasource 'CartoDB:pramsey tables=publicart',
    format 'CartoDB' );

CREATE FOREIGN TABLE publicart (
  fid integer,
  the_geom geometry,
  objectid varchar,
  location varchar,
  installed varchar,
  artproject varchar,
  title varchar,
  artist varchar
  SERVER cartodb
  OPTIONS ( layer 'publicart' );
```

Here’s the same thing, only using CartoDB as the foreign server.
Even though CartoDB is PostgreSQL/PostGIS underneath, we don’t have access to the low level, so we define our server using the CartoDB OGR driver type rather than the PostgreSQL OGR driver.
Then we define our foreign table to match the CartoDB table.

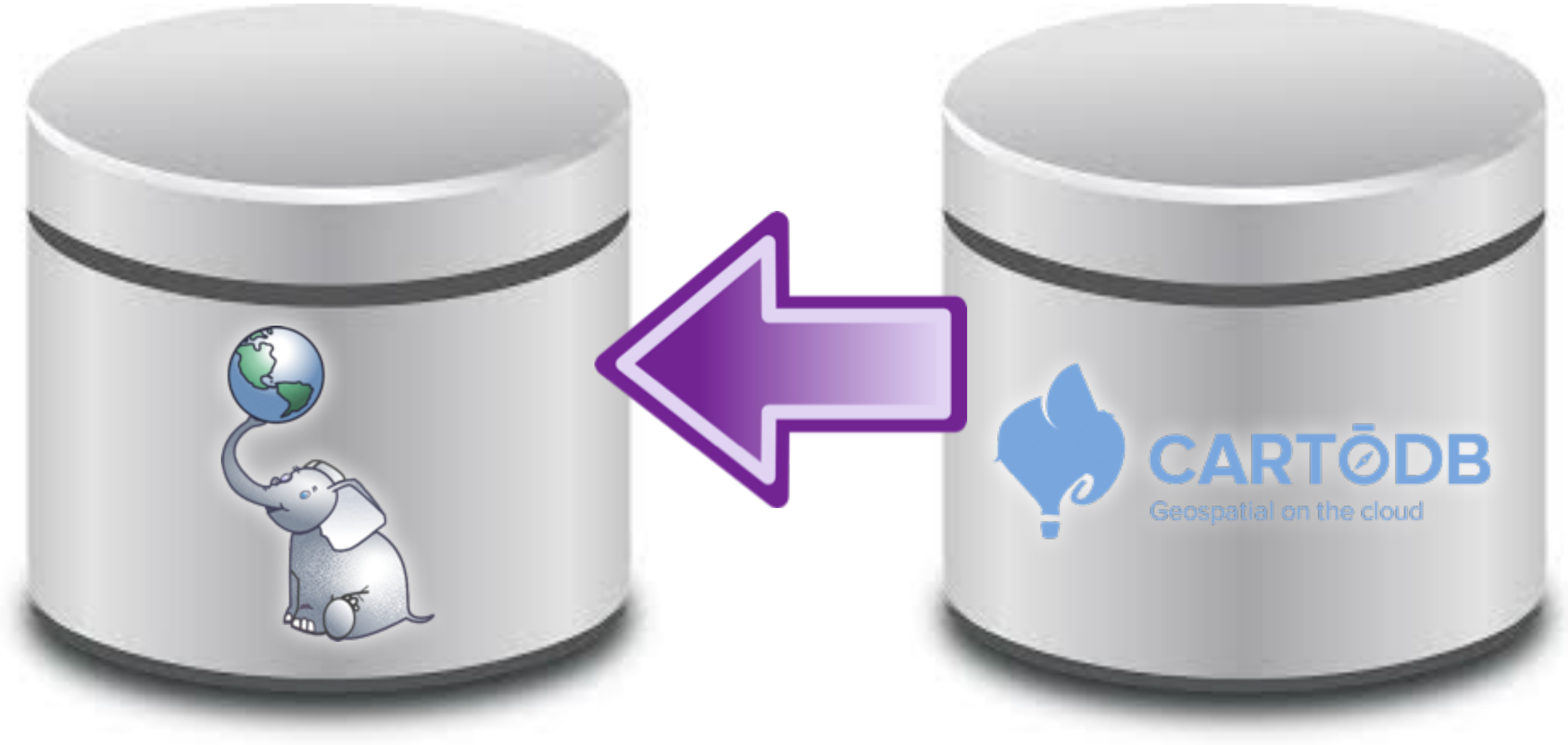


```
SELECT
  ST_Distance_Sphere(a.the_geom, b.the_geom),
  b.title
FROM publicart a, publicart b
WHERE a.title = 'Fire in the Belly'
ORDER BY st_distance_sphere
ASC LIMIT 7;
```

```
--------------------+------------------------------------
 st_distance_sphere |                title
--------------------+------------------------------------
                  0 | Fire in the Belly
       26.169185324 | Untitled
       57.207071845 | Park Dreams
       66.483303836 | Asymmetric Beauty
      128.499756793 | Peanut Vendor's Memory
       253.83576748 | Trust & Harmony
      268.699036353 | Pembroke Plaza Public Art Mosaic
```

Once it is defined, we can run queries locally on the table, and get results just as if the data were local.
Here's a distance query, finding the 7 nearest pieces of public art to the piece named "fire in the belly"

data synchronization
“pushdowns”



The OGR FDW driver is getting better all the time. It can now send “quals”, that is, WHERE clauses to the remote servers, so that only subsets of the data are sent back to the client. Next up it will support spatial filters, and then finally UPDATE and DELETE queries, so it’s possible to edit the remote data without ever leaving the friendly confines of your local PostgreSQL instance.

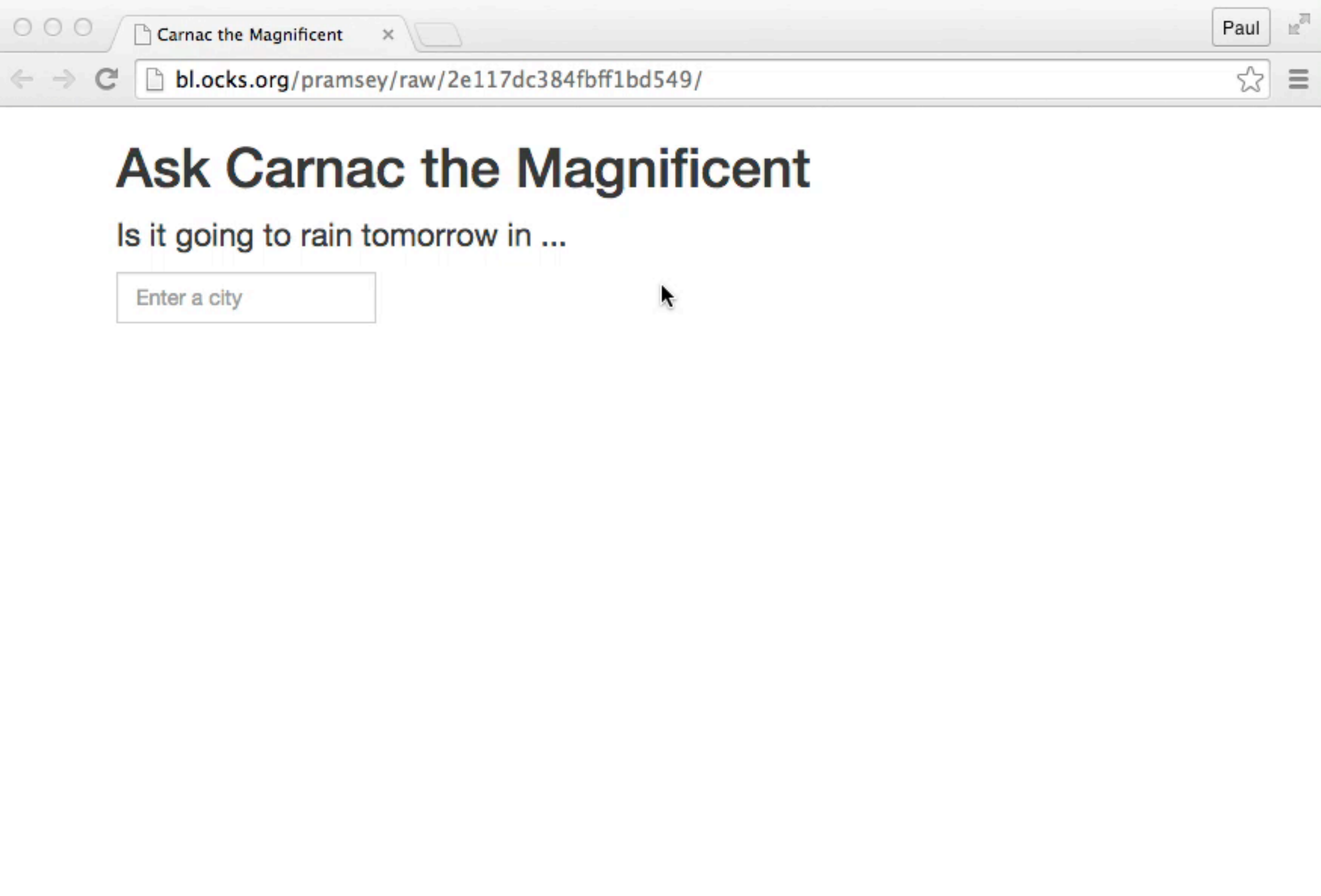
3rd movement

time and tide

I think I’m going to have to give a talk modeled on Vivaldi’s four seasons some time... summer fall winter spring, that would be cool.... for now the movement names are a little arbitrary...

“look into the future”

in my abstract I may have promised that in this talk I would show how to use PostGIS to “look into the future” so, for this section I want to turn it over to Carnac, the Magnificent



Ask Carnac the Magnificent

Is it going to rain tomorrow in ...

Enter a city

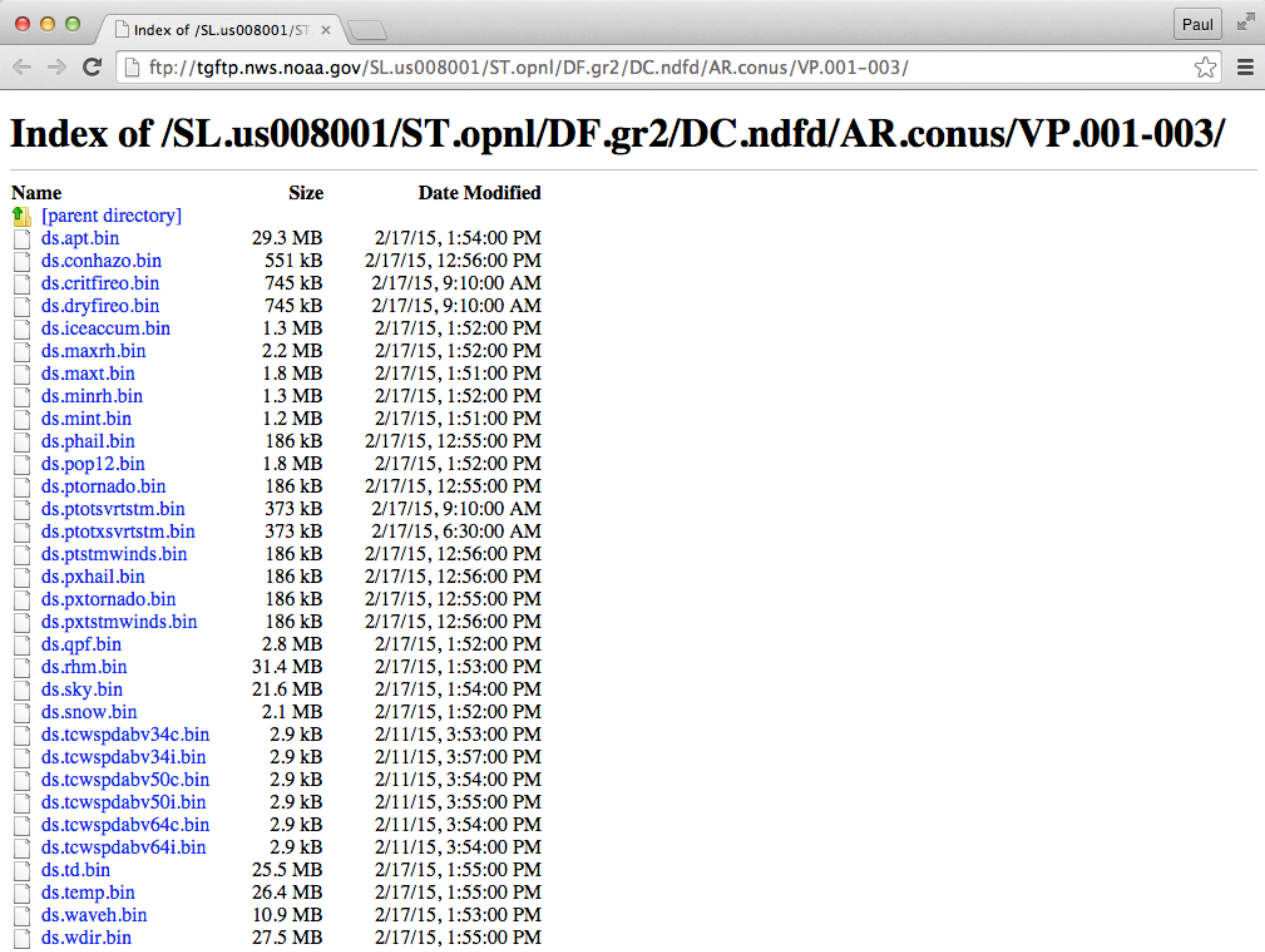
Demo http://bl.ocks.org/pramsey

so this is what carnac does... you type a city into the autocomplete form
(which is driven off of a PostgreSQL query, of course)
and based on the city, Carnac tells you if it's going to rain "tomorrow"

All done with PostgreSQL and PostGIS!

The video was taken a couple weeks ago, so the answers there are wrong,
but if you go to the demo page you can find a live Carnac that should be
pretty accurate
(or as accurate as a fortune teller could be expected to be)

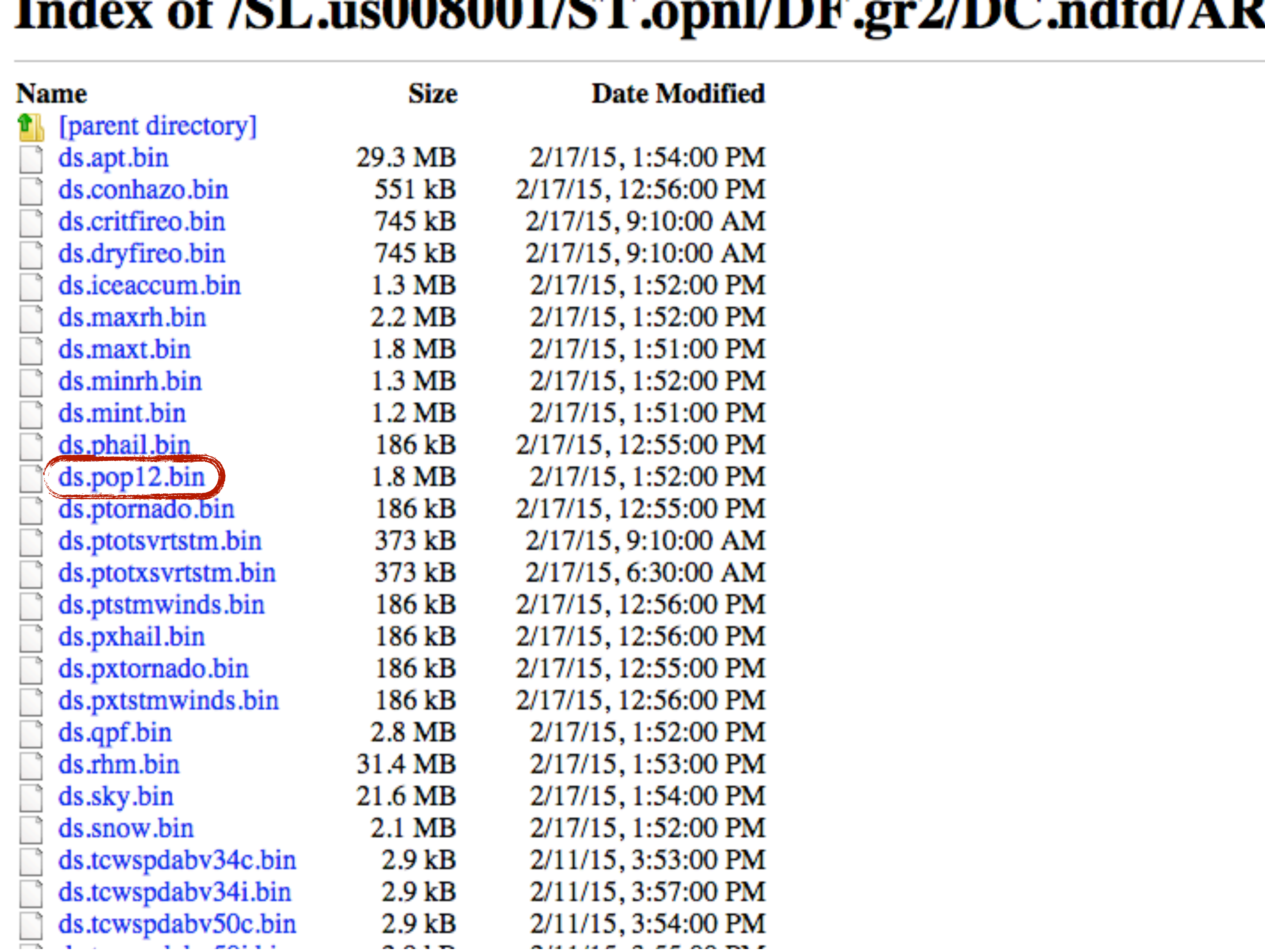
So, let's peel back the covers,
and see how this trick works


Index of /SL.us008001/ST
Paul
ftp://tgftp.nws.noaa.gov/SL.us008001/ST.opnl/DF.gr2/DC.ndfd/AR.conus/VP.001-003/

Index of /SL.us008001/ST.opnl/DF.gr2/DC.ndfd/AR.conus/VP.001-003/

Name	Size	Date Modified
[parent directory]		
ds.apt.bin	29.3 MB	2/17/15, 1:54:00 PM
ds.conhazo.bin	551 kB	2/17/15, 12:56:00 PM
ds.critfireo.bin	745 kB	2/17/15, 9:10:00 AM
ds.dryfireo.bin	745 kB	2/17/15, 9:10:00 AM
ds.iceaccum.bin	1.3 MB	2/17/15, 1:52:00 PM
ds.maxrh.bin	2.2 MB	2/17/15, 1:52:00 PM
ds.maxt.bin	1.8 MB	2/17/15, 1:51:00 PM
ds.minrh.bin	1.3 MB	2/17/15, 1:52:00 PM
ds.mint.bin	1.2 MB	2/17/15, 1:51:00 PM
ds.phail.bin	186 kB	2/17/15, 12:55:00 PM
ds.pop12.bin	1.8 MB	2/17/15, 1:52:00 PM
ds.ptornado.bin	186 kB	2/17/15, 12:55:00 PM
ds.ptotsvrtstm.bin	373 kB	2/17/15, 9:10:00 AM
ds.ptotxsvrtstm.bin	373 kB	2/17/15, 6:30:00 AM
ds.ptstmwinds.bin	186 kB	2/17/15, 12:56:00 PM
ds.pxhail.bin	186 kB	2/17/15, 12:56:00 PM
ds.pxtornado.bin	186 kB	2/17/15, 12:55:00 PM
ds.pxtstmwinds.bin	186 kB	2/17/15, 12:56:00 PM
ds.qpf.bin	2.8 MB	2/17/15, 1:52:00 PM
ds.rhm.bin	31.4 MB	2/17/15, 1:53:00 PM
ds.sky.bin	21.6 MB	2/17/15, 1:54:00 PM
ds.snow.bin	2.1 MB	2/17/15, 1:52:00 PM
ds.tcwspdabv34c.bin	2.9 kB	2/11/15, 3:53:00 PM
ds.tcwspdabv34i.bin	2.9 kB	2/11/15, 3:57:00 PM
ds.tcwspdabv50c.bin	2.9 kB	2/11/15, 3:54:00 PM
ds.tcwspdabv50i.bin	2.9 kB	2/11/15, 3:55:00 PM
ds.tcwspdabv64c.bin	2.9 kB	2/11/15, 3:54:00 PM
ds.tcwspdabv64i.bin	2.9 kB	2/11/15, 3:54:00 PM
ds.td.bin	25.5 MB	2/17/15, 1:55:00 PM
ds.temp.bin	26.4 MB	2/17/15, 1:55:00 PM
ds.waveh.bin	10.9 MB	2/17/15, 1:53:00 PM
ds.wdir.bin	27.5 MB	2/17/15, 1:55:00 PM

NOAA is nice enough to publish their forecast data, on a web directory that is kept constantly up to date (so you can see when I created this slide, Feb 27) and for the rain prediction

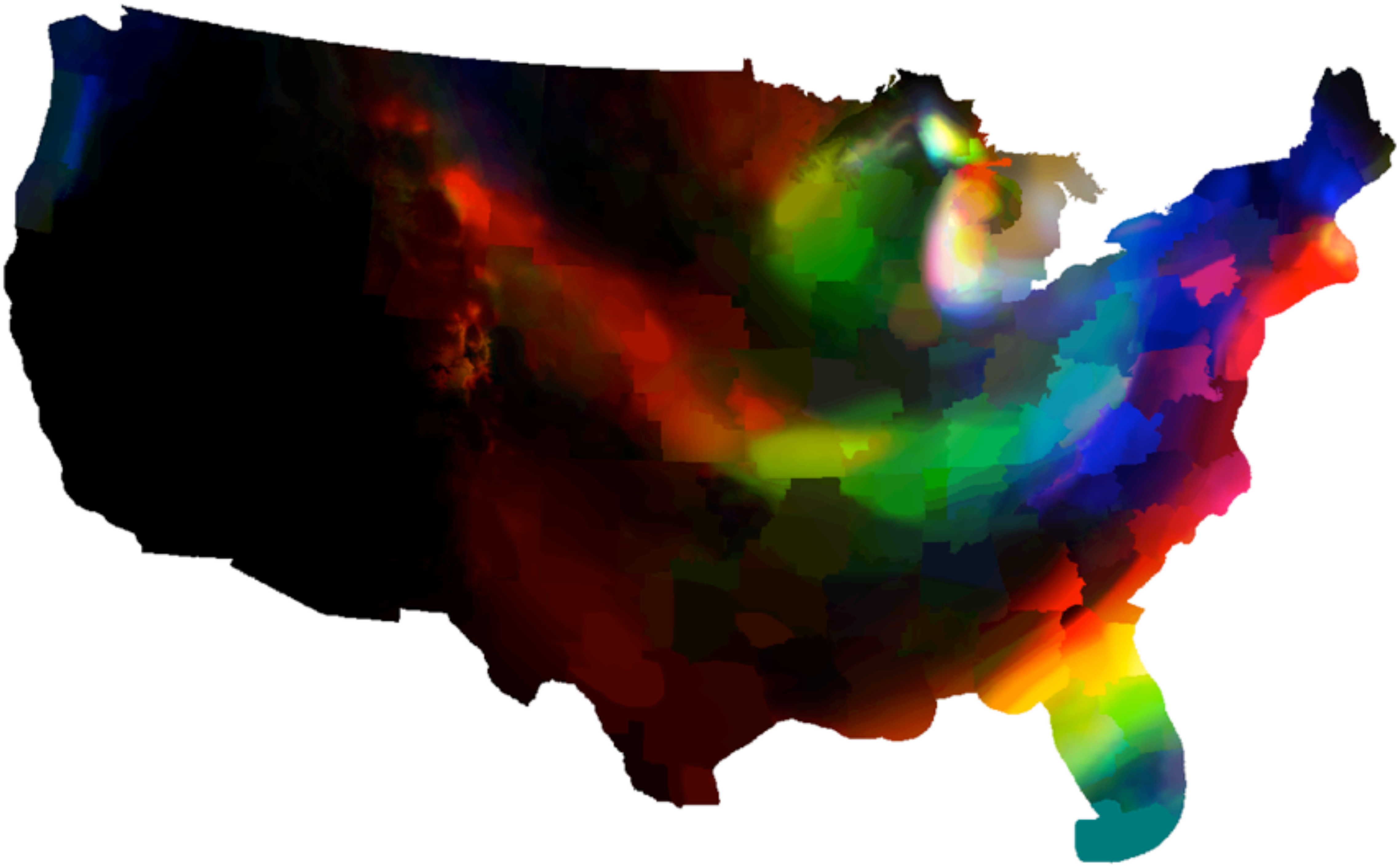


Index of /SL.us008001/ST.opnl/DF.gr2/DC.ndfd/AR

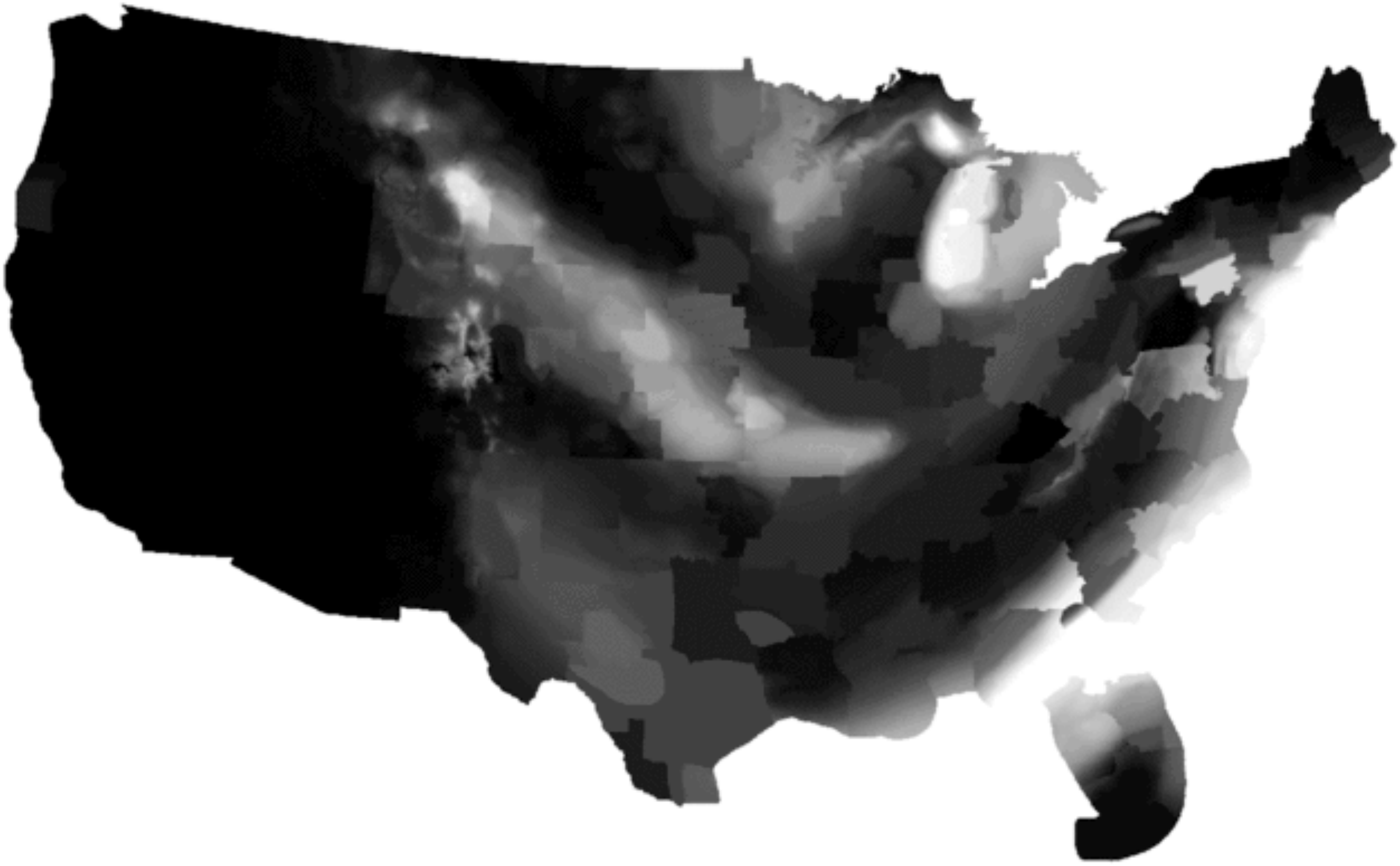
Name	Size	Date Modified
[parent directory]		
ds.apt.bin	29.3 MB	2/17/15, 1:54:00 PM
ds.conhazo.bin	551 kB	2/17/15, 12:56:00 PM
ds.critfireo.bin	745 kB	2/17/15, 9:10:00 AM
ds.dryfireo.bin	745 kB	2/17/15, 9:10:00 AM
ds.iceaccum.bin	1.3 MB	2/17/15, 1:52:00 PM
ds.maxrh.bin	2.2 MB	2/17/15, 1:52:00 PM
ds.maxt.bin	1.8 MB	2/17/15, 1:51:00 PM
ds.minrh.bin	1.3 MB	2/17/15, 1:52:00 PM
ds.mint.bin	1.2 MB	2/17/15, 1:51:00 PM
ds.phail.bin	186 kB	2/17/15, 12:55:00 PM
ds.pop12.bin	1.8 MB	2/17/15, 1:52:00 PM
ds.ptornado.bin	186 kB	2/17/15, 12:55:00 PM
ds.ptotsvrtstm.bin	373 kB	2/17/15, 9:10:00 AM
ds.ptotxsvrtstm.bin	373 kB	2/17/15, 6:30:00 AM
ds.ptstmwinds.bin	186 kB	2/17/15, 12:56:00 PM
ds.pxhail.bin	186 kB	2/17/15, 12:56:00 PM
ds.pxtornado.bin	186 kB	2/17/15, 12:55:00 PM
ds.pxtstmwinds.bin	186 kB	2/17/15, 12:56:00 PM
ds.qpf.bin	2.8 MB	2/17/15, 1:52:00 PM
ds.rhm.bin	31.4 MB	2/17/15, 1:53:00 PM
ds.sky.bin	21.6 MB	2/17/15, 1:54:00 PM
ds.snow.bin	2.1 MB	2/17/15, 1:52:00 PM
ds.tcwspdabv34c.bin	2.9 kB	2/11/15, 3:53:00 PM
ds.tcwspdabv34i.bin	2.9 kB	2/11/15, 3:57:00 PM
ds.tcwspdabv50c.bin	2.9 kB	2/11/15, 3:54:00 PM

the file we are interested in is the "pop12" file, the probability of precipitation given in 12 hour forecast windows

if you download the file and convert it to a GeoTIFF and look at it in QGIS, this is what you see



Which is totally awesome and trippy!!
QGIS defaults to loading the TIFF with Band 1 as Red, Band 2 as Green and Band 3 as Blue, which gives this really cool picture with lots of fun mixing.
Actually, each band is meant to be viewed separately as a forecast period.



And if you string them together
you can see the forecast pattern of precipitation moving west to east,
as one would expect given the general direction of weather and the jet stream in North America.

So, once the data are in GeoTIFF, we can use them in PostGIS, But...

```
gdal_translate
  ds.pop12.bin pop12.tif
```

actually,
getting an optimal conversion from the NOAA NetCDF format into GeoTIFF
using GDAL was a bit of an adventure and a learning experience,

The default conversion did preserve all five bands, and included the spatial reference information and GRIB metadata about the input file, which was great,
BUT the input file was 1.5Mb and the output file was 113Mb!

So, the first thing you notice when you pop open the output file is that the pixel type is double,
so that’s 8 bytes per pixel, but the input data is just integers from 0-100 and nodata at 9999, which basically fits into a single byte. So there’s an 8-fold improvement in storage available if we just change the pixel type

```
gdal_translate
  -ot Byte
  ds.pop12.bin pop12.tif
```

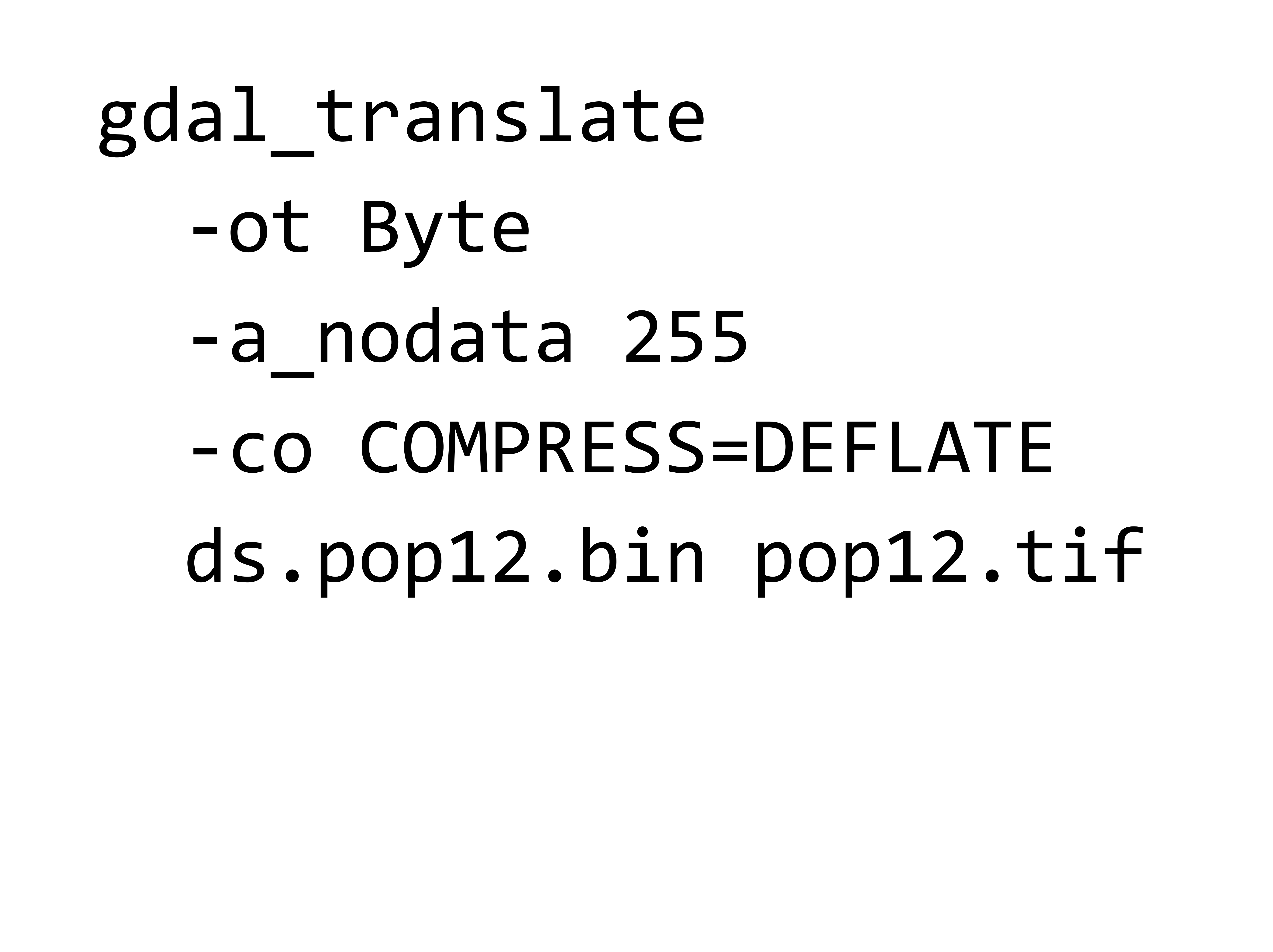
Which is pretty easy,
and that gets the output file down to 14mb,
so no longer 100 times larger than the input, only 10 times larger
which is still pretty terrible
and when you look at the tiff in qgis, it has these awful imperfections, where the 9999 NODATA pixels have been coerced down into the same range as the data from 0-100

```
gdal_translate
   -ot Byte
   -a_nodata 255
   ds.pop12.bin pop12.tif
```

So we need to explicitly map the nodata values into a slot in the number space where there's no real data

Since our data run from 0-100, we can map it into 255 safely

But the file is still large, what's going on?



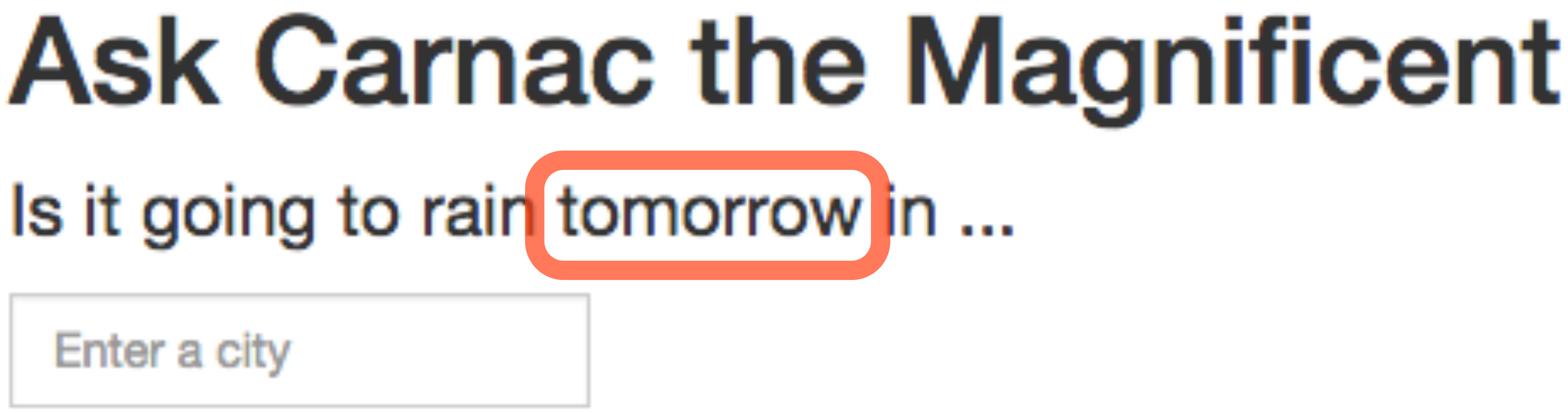
```
gdal_translate
   -ot Byte
   -a_nodata 255
   -co COMPRESS=DEFLATE
   ds.pop12.bin pop12.tif
```

Well, it turns out that GDAL is producing an UNCOMPRESSED geotiff by default

So we have to ASK for compression
Deflate does an excellent job, and now we’re down to 1.4Mb, about the same as the original
But it turns out that DEFLATE has some extra options to twiddle

```
gdal_translate
  -ot Byte
  -a_nodata 255
  -co COMPRESS=DEFLATE
  -co ZLEVEL=9
  -co PREDICTOR=2
  ds.pop12.bin pop12.tif
```

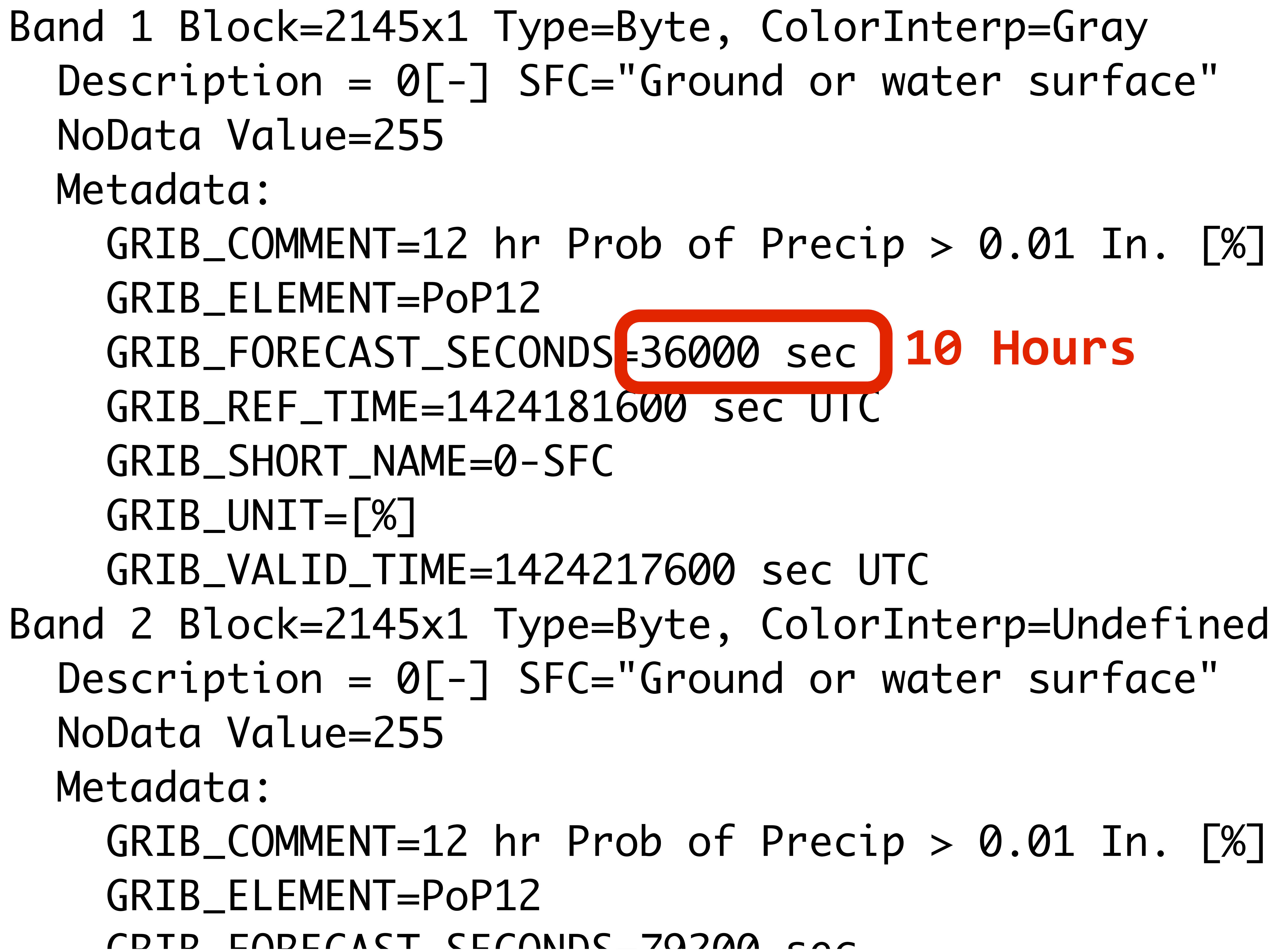
And if we add a higher compression level
(which uses slightly more RAM, something we don’t mind)
and we add a scan-line predictor
(which makes sense since our data tend to be spatially autocorrelated)
we can get down to just over 1Mb!
which is a **pretty nice improvement** over the 113Mb the default conversion gave us



Ask Carnac the Magnificent

Is it going to rain tomorrow in ...

so, a key component of our solution is what exactly Carnac
means when he says “tomorrow”
and to figure that out, it helps to look at the output from gdalinfo



```
Band 1 Block=2145x1 Type=Byte, ColorInterp=Gray
  Description = 0[-] SFC="Ground or water surface"
  NoData Value=255
  Metadata:
    GRIB_COMMENT=12 hr Prob of Precip > 0.01 In. [%]
    GRIB_ELEMENT=PoP12
    GRIB_FORECAST_SECONDS=36000 sec
    GRIB_REF_TIME=1424181600 sec UTC
    GRIB_SHORT_NAME=0-SFC
    GRIB_UNIT=[%]
    GRIB_VALID_TIME=1424217600 sec UTC
Band 2 Block=2145x1 Type=Byte, ColorInterp=Undefined
  Description = 0[-] SFC="Ground or water surface"
  NoData Value=255
  Metadata:
    GRIB_COMMENT=12 hr Prob of Precip > 0.01 In. [%]
    GRIB_ELEMENT=PoP12
    GRIB_FORECAST_SECONDS=79200 sec
```

10 Hours

There’s actually 5 bands in the GeoTIFF,
and GDAL does a great job preserving the original GRIB format metadata
so we can figure out what each band MEANS.
The first band is good for 10 hours after the forecast is generated.

```
    GRIB_VALID_TIME=1424217600 sec UTC
Band 2 Block=2145x1 Type=Byte, ColorInterp=Undefined
  Description = 0[-] SFC="Ground or water surface"
  NoData Value=255
  Metadata:
    GRIB_COMMENT=12 hr Prob of Precip > 0.01 In. [%]
    GRIB_ELEMENT=PoP12
    GRIB_FORECAST_SECONDS=79200 sec
    GRIB_REF_TIME=1424181600 sec UTC
    GRIB_SHORT_NAME=0-SFC
    GRIB_UNIT=[%]
    GRIB_VALID_TIME=1424260800 sec UTC
Band 3 Block=2145x1 Type=Byte, ColorInterp=Undefined
  Description = 0[-] SFC="Ground or water surface"
  NoData Value=255
  Metadata:
    GRIB_COMMENT=12 hr Prob of Precip > 0.01 In. [%]
    GRIB_ELEMENT=PoP12
    GRIB_FORECAST_SECONDS=122400 sec
```

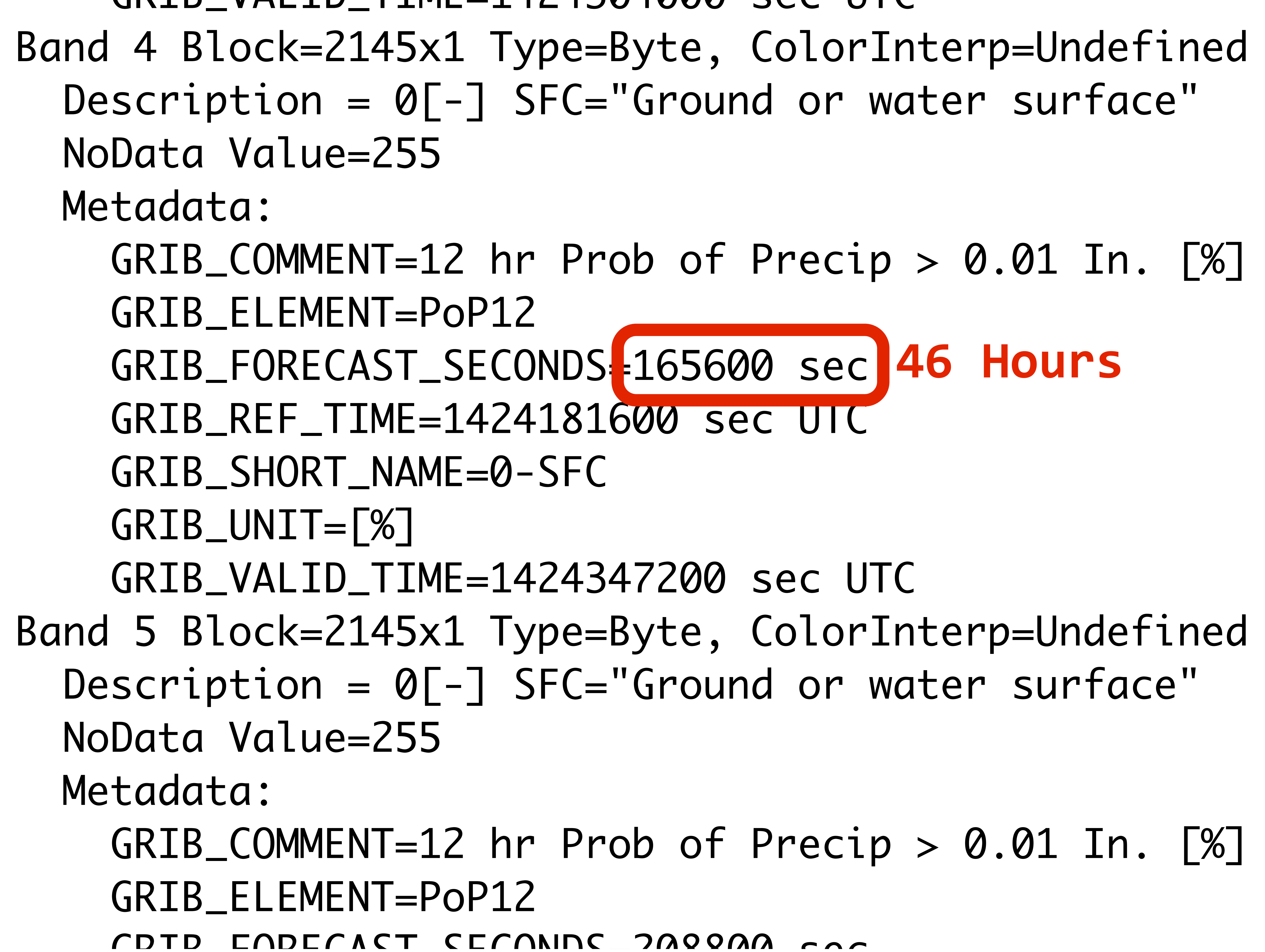
22 Hours

The second band is good until 22 hours from the forecast

```
    GRIB_VALID_TIME=1424260800 sec UTC
Band 3 Block=2145x1 Type=Byte, ColorInterp=Undefined
  Description = 0[-] SFC="Ground or water surface"
  NoData Value=255
  Metadata:
    GRIB_COMMENT=12 hr Prob of Precip > 0.01 In. [%]
    GRIB_ELEMENT=PoP12
    GRIB_FORECAST_SECONDS=122400 sec
    GRIB_REF_TIME=1424181600 sec UTC
    GRIB_SHORT_NAME=0-SFC
    GRIB_UNIT=[%]
    GRIB_VALID_TIME=1424304000 sec UTC
Band 4 Block=2145x1 Type=Byte, ColorInterp=Undefined
  Description = 0[-] SFC="Ground or water surface"
  NoData Value=255
  Metadata:
    GRIB_COMMENT=12 hr Prob of Precip > 0.01 In. [%]
    GRIB_ELEMENT=PoP12
    GRIB_FORECAST_SECONDS=165600 sec
```

34 Hours

The third band is good until 34 hours from the forecast



```
    GRIB_VALID_TIME=1424304000 sec UTC
Band 4 Block=2145x1 Type=Byte, ColorInterp=Undefined
  Description = 0[-] SFC="Ground or water surface"
  NoData Value=255
  Metadata:
    GRIB_COMMENT=12 hr Prob of Precip > 0.01 In. [%]
    GRIB_ELEMENT=PoP12
    GRIB_FORECAST_SECONDS=165600 sec
    GRIB_REF_TIME=1424181600 sec UTC
    GRIB_SHORT_NAME=0-SFC
    GRIB_UNIT=[%]
    GRIB_VALID_TIME=1424347200 sec UTC
Band 5 Block=2145x1 Type=Byte, ColorInterp=Undefined
  Description = 0[-] SFC="Ground or water surface"
  NoData Value=255
  Metadata:
    GRIB_COMMENT=12 hr Prob of Precip > 0.01 In. [%]
    GRIB_ELEMENT=PoP12
    GRIB_FORECAST_SECONDS=208800 sec
```

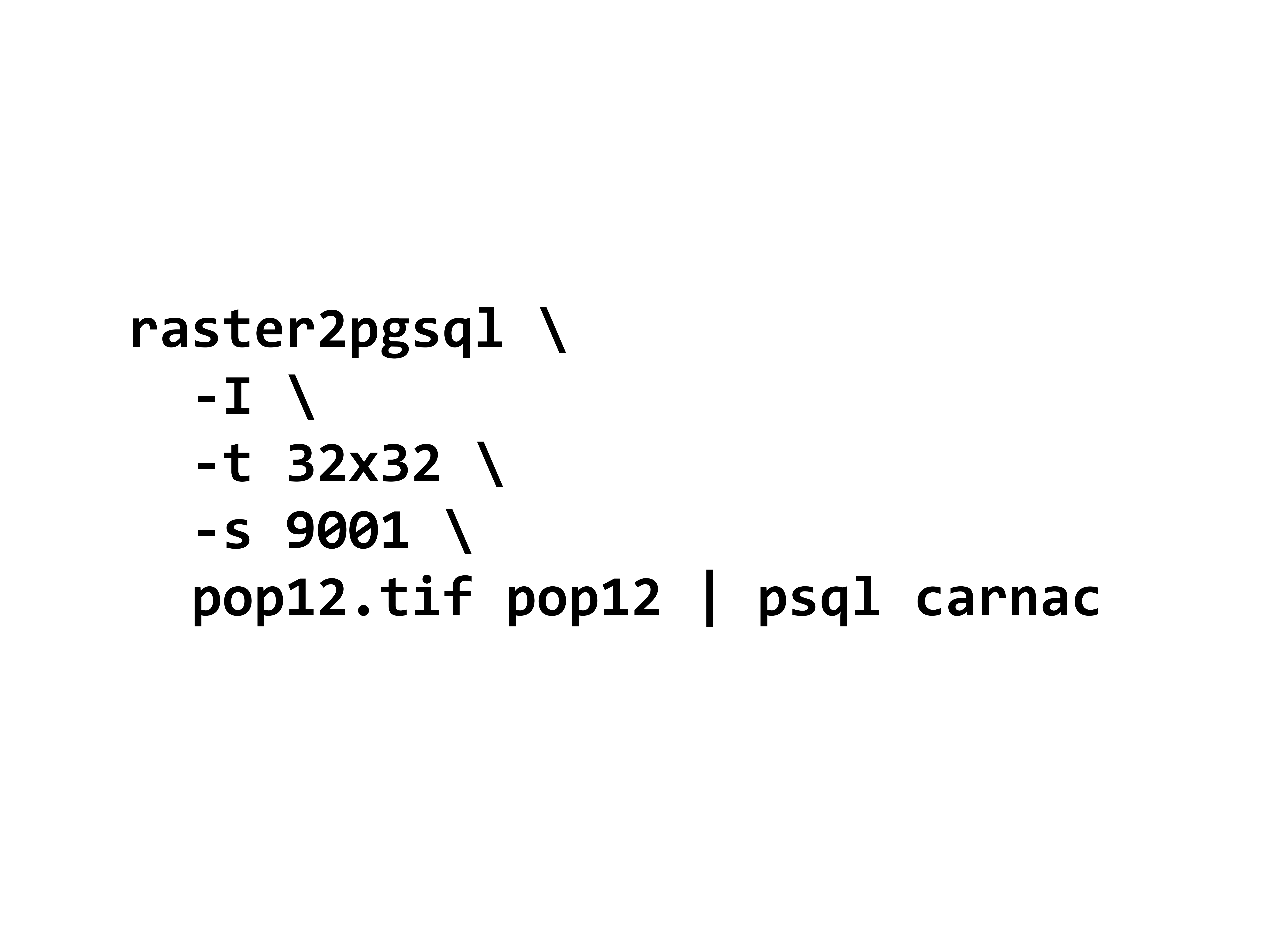
46 Hours

The fourth band is good until 46 hours from the forecast
Each band also has a “valid time” which gives the UTC timestamp
when the forecast expires, that could be pretty useful...

ideal solution: check the valid times of bands and use that to figure out when “tomorrow” is, relative to now

my solution: just use bands 3 and 4! could also use bands 2 and 3, matter of interpretation, decided to go with more “future” biased hack

There are actually two ways to solve the problem,
the right way, which would carefully look at the GRIB metadata to figure out what forecast band we needed to use
and *my* way, which was to just average bands 3 and 4 together.



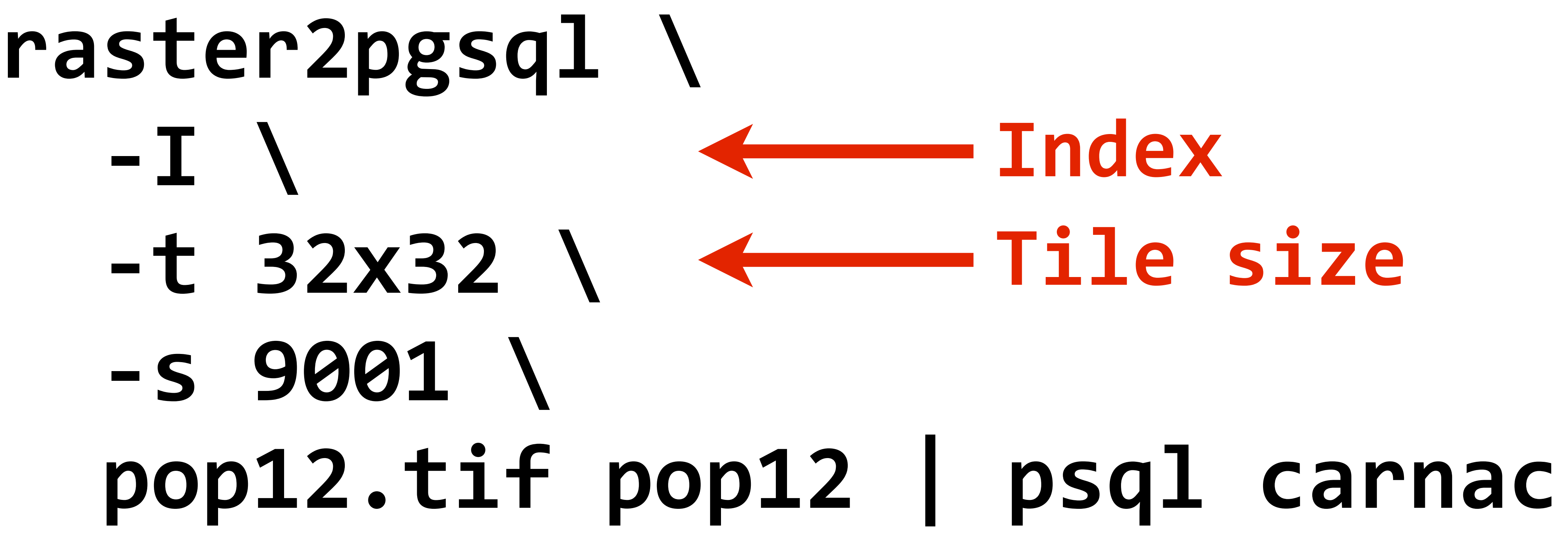
```
raster2pgsql \
  -I \
  -t 32x32 \
  -s 9001 \
  pop12.tif pop12 | psql carnac
```

So, before I can do any averaging, first I needed to load the data,
which involves the usual opaque command-line syntax, but there’s nothing
too crazy here,
we choose 32x32 chips because a 1 bytes pixel times 32 times 32 times 5 bands implies a 5k tile, which is slightly smaller than the 8k page size of PostgreSQL
Once all the data are loaded, we just need two SQL queries to run the magic of Carnac,

```
raster2pgsql \
  -I \
  -t 32x32 \
  -s 9001 \
  pop12.tif pop12 | psql carnac
```

Index

So, before I can do any averaging, first I needed to load the data,
which involves the usual opaque command-line syntax, but there’s nothing
too crazy here,
we choose 32x32 chips because a 1 bytes pixel times 32 times 32 times 5 bands implies a 5k
tile, which is slightly smaller than the 8k page size of PostgreSQL
Once all the data are loaded, we just need two SQL queries to run the magic of Carnac,

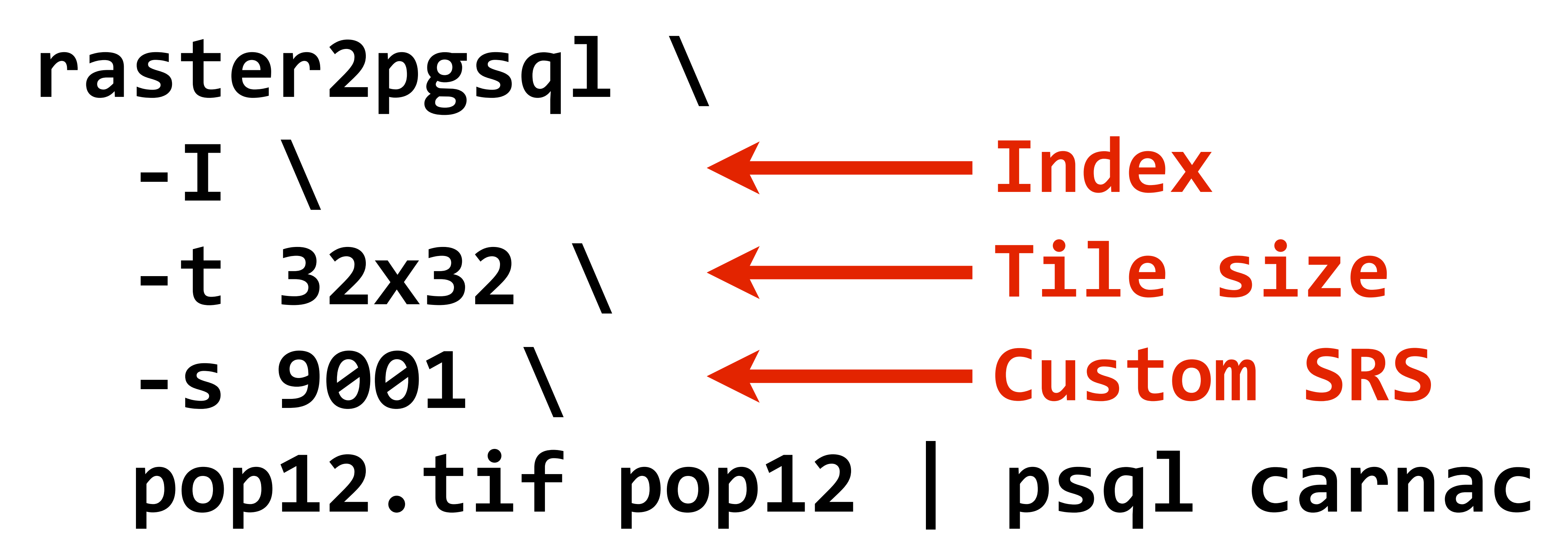


```
raster2pgsql \
  -I \
  -t 32x32 \
  -s 9001 \
  pop12.tif pop12 | psql carnac
```

Index

Tile size

So, before I can do any averaging, first I needed to load the data,
which involves the usual opaque command-line syntax, but there’s nothing
too crazy here,
we choose 32x32 chips because a 1 bytes pixel times 32 times 32 times 5 bands implies a 5k tile, which is slightly smaller than the 8k page size of PostgreSQL
Once all the data are loaded, we just need two SQL queries to run the magic of Carnac,



```
raster2pgsql \
  -I \
  -t 32x32 \
  -s 9001 \
  pop12.tif pop12 | psql carnac
```

Index

Tile size

Custom SRS

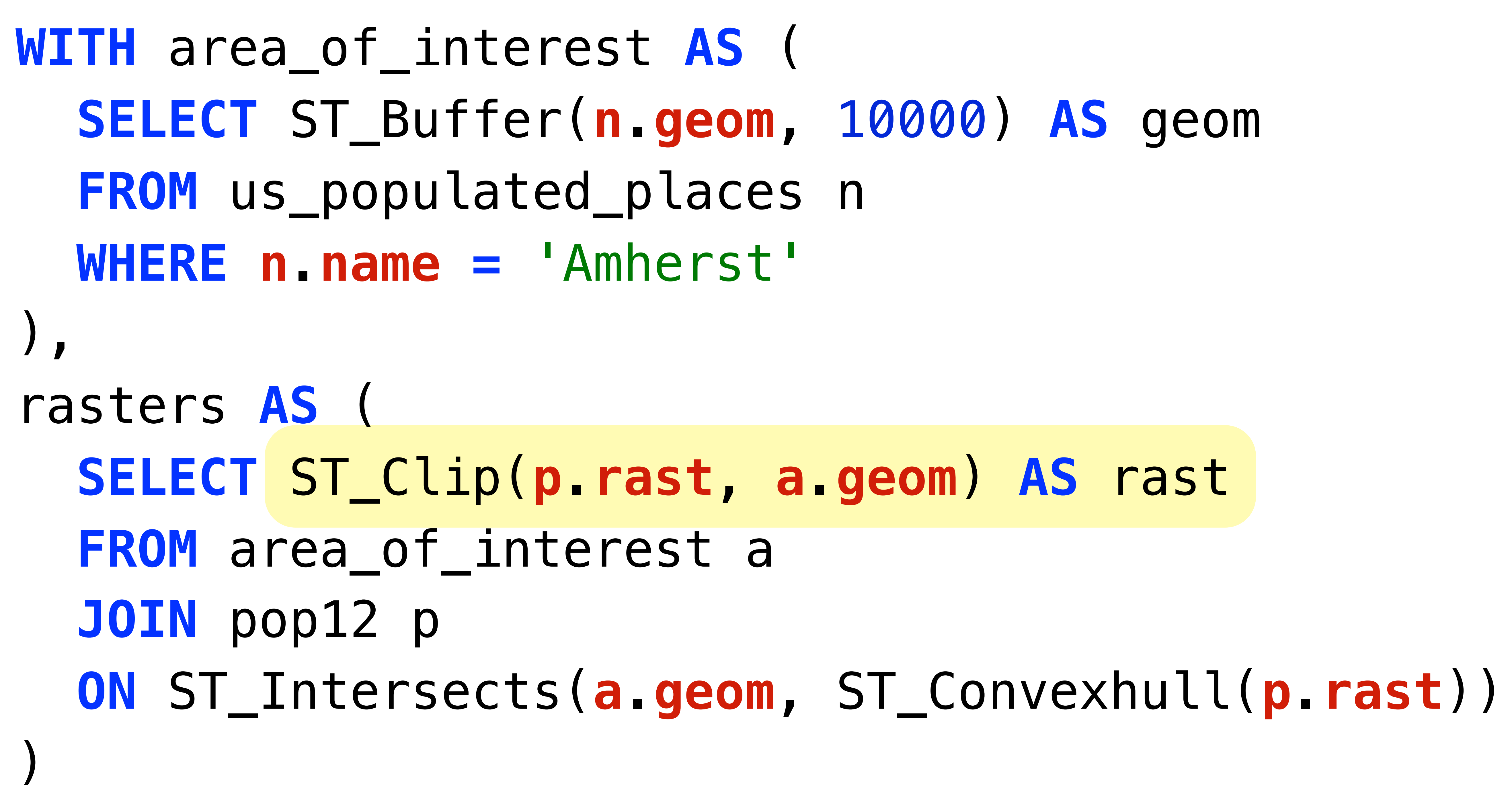
So, before I can do any averaging, first I needed to load the data,
which involves the usual opaque command-line syntax, but there’s nothing too crazy here,
we choose 32x32 chips because a 1 bytes pixel times 32 times 32 times 5 bands implies a 5k tile, which is slightly smaller than the 8k page size of PostgreSQL
Once all the data are loaded, we just need two SQL queries to run the magic of Carnac,

```
SELECT
   name,
   cartodb_id,
   population
FROM us_populated_places
WHERE name ILIKE 'gr%'
ORDER BY population
DESC LIMIT 10;
```

First, the query to drive the autocomplete form,
which just reads the table of populated places, ordering the results by size of city.

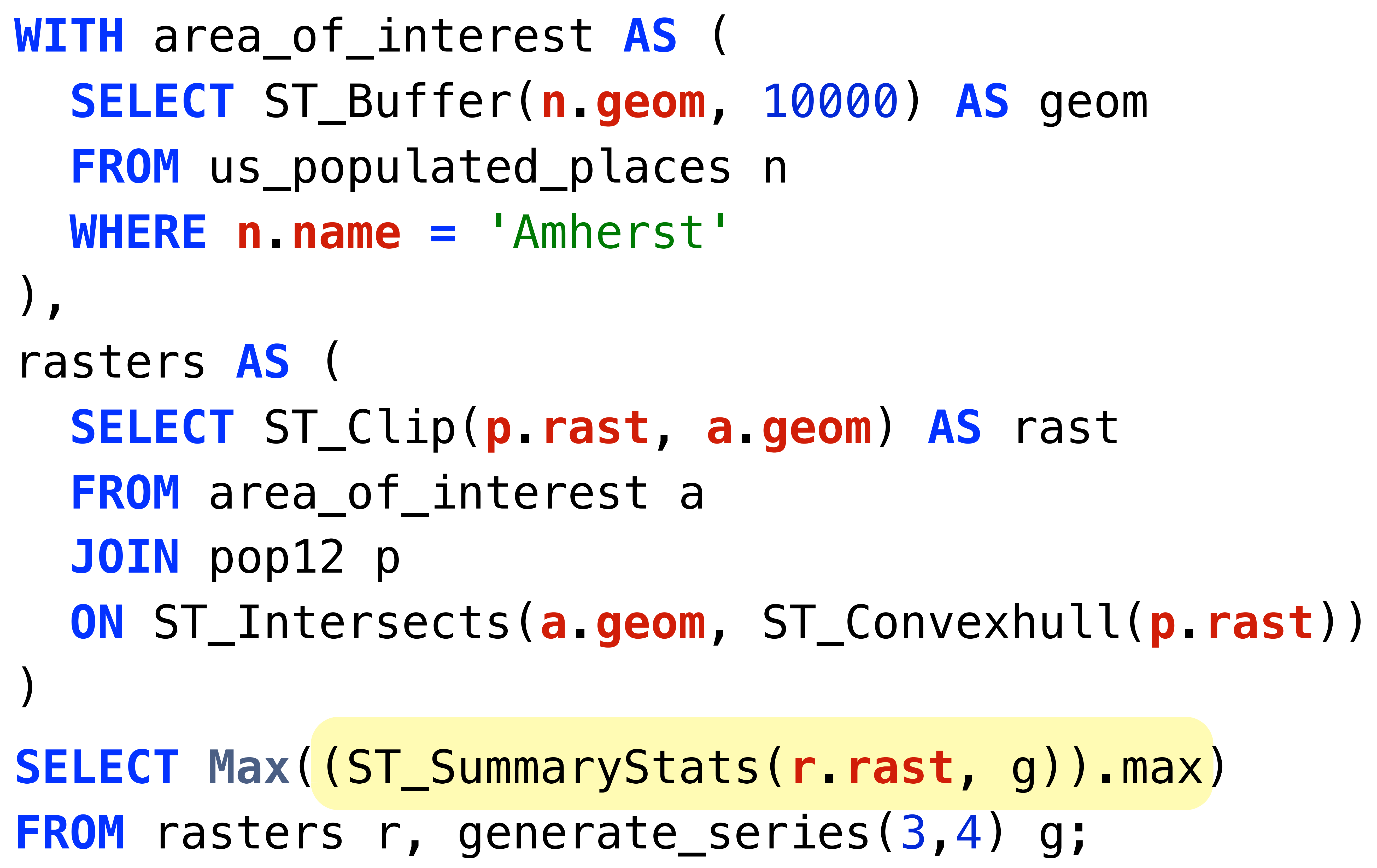
```
WITH area_of_interest AS (
  SELECT ST_Buffer(n.geom, 10000) AS geom
  FROM us_populated_places n
  WHERE n.name = 'Amherst'
)
```

Second, the query that generates the probability guess for Carnac, given the city.
So, step one, we use the selected city to generate a buffer ring that we will use to summarize the precipitation probabilities.
<x> Step two, we apply that buffer to the raster table, and find all the rasters that intersect the buffer, and then masked out just the pixels of those rasters that fall WITHIN the buffer.
<x> Step three, we take those masked rasters and summarize them, finding the maximum value of the probability of precipitation for every pixel. That becomes the number we use to drive Carnac’s guess.



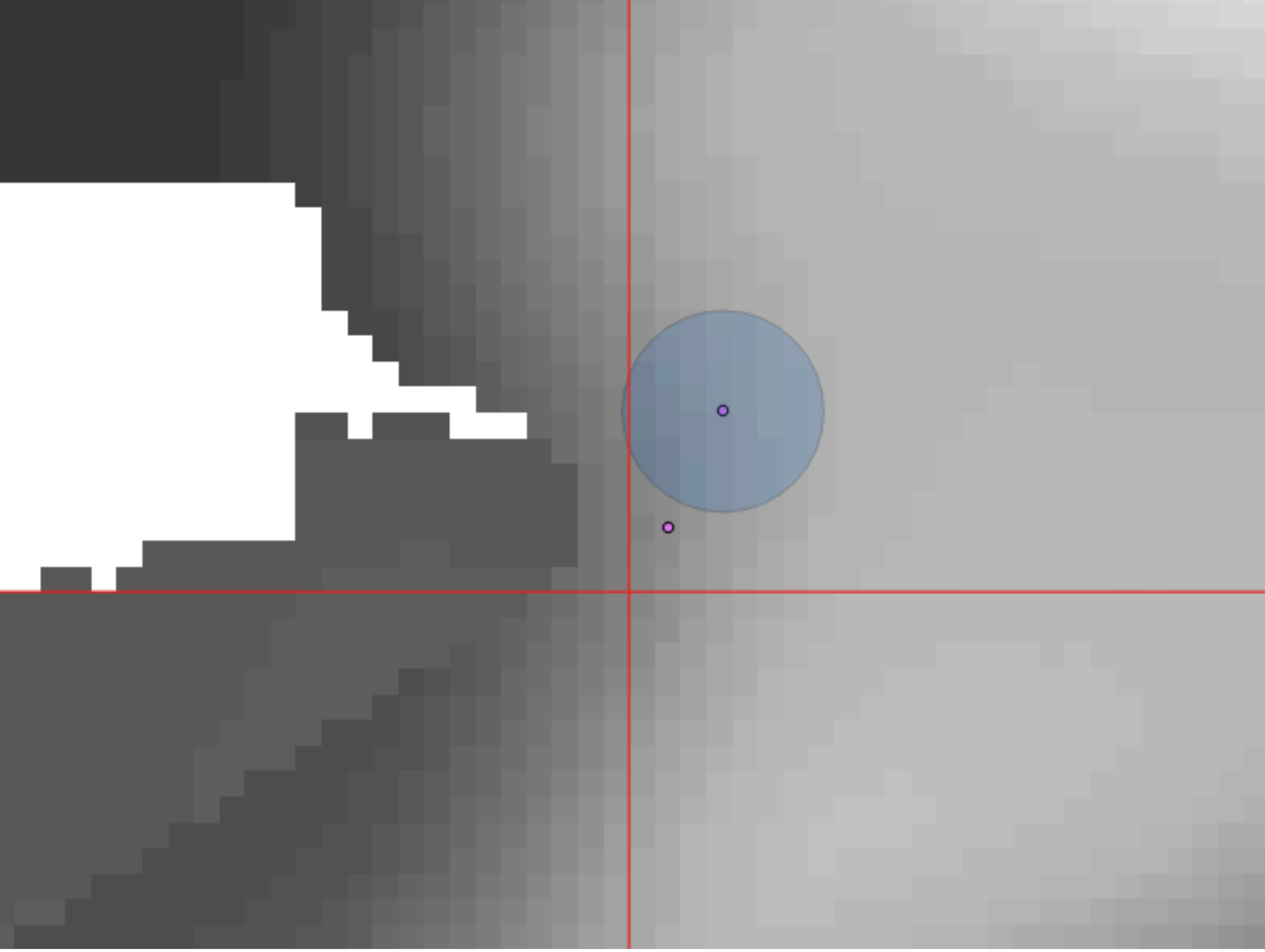
```
WITH area_of_interest AS (
  SELECT ST_Buffer(n.geom, 10000) AS geom
  FROM us_populated_places n
  WHERE n.name = 'Amherst'
),
rasters AS (
  SELECT ST_Clip(p.rast, a.geom) AS rast
  FROM area_of_interest a
  JOIN pop12 p
  ON ST_Intersects(a.geom, ST_Convexhull(p.rast))
)
```

Second, the query that generates the probability guess for Carnac, given the city.
So, step one, we use the selected city to generate a buffer ring that we will use to summarize the precipitation probabilities.
<x> Step two, we apply that buffer to the raster table, and find all the rasters that intersect the buffer, and then masked out just the pixels of those rasters that fall WITHIN the buffer.
<x> Step three, we take those masked rasters and summarize them, finding the maximum value of the probability of precipitation for every pixel. That becomes the number we use to drive Carnac’s guess.



```
WITH area_of_interest AS (
  SELECT ST_Buffer(n.geom, 10000) AS geom
  FROM us_populated_places n
  WHERE n.name = 'Amherst'
),
rasters AS (
  SELECT ST_Clip(p.rast, a.geom) AS rast
  FROM area_of_interest a
  JOIN pop12 p
  ON ST_Intersects(a.geom, ST_Convexhull(p.rast))
)
SELECT Max((ST_SummaryStats(r.rast, g)).max)
FROM rasters r, generate_series(3,4) g;
```

Second, the query that generates the probability guess for Carnac, given the city.
So, step one, we use the selected city to generate a buffer ring that we will use to summarize the precipitation probabilities.
<x> Step two, we apply that buffer to the raster table, and find all the rasters that intersect the buffer, and then masked out just the pixels of those rasters that fall WITHIN the buffer.
<x> Step three, we take those masked rasters and summarize them, finding the maximum value of the probability of precipitation for every pixel. That becomes the number we use to drive Carnac's guess.



Here’s what it looks like visually.
Calculate the blue buffer,
Use the buffer to find the intersecting chips (there’s two of them, barely)
and then mask out the chips to just find the pixels that intersect the buffer.

```
#!/bin/bash

cd /tmp
url=ftp://tgftp.nws.noaa.gov/SL.us008001/ST.opnl/
DF.gr2/DC.ndfd/AR.conus/VP.001-003/ds.pop12.bin
wget $url

gdal_translate -q \
  -a_nodata 255 \
  -co ZLEVEL=9 \
  -co PREDICTOR=2 \
  -co COMPRESS=DEFLATE \
  -ot Byte \
  ds.pop12.bin pop12.tif

$HOME/s3upload/upload.sh pop12.tif
```

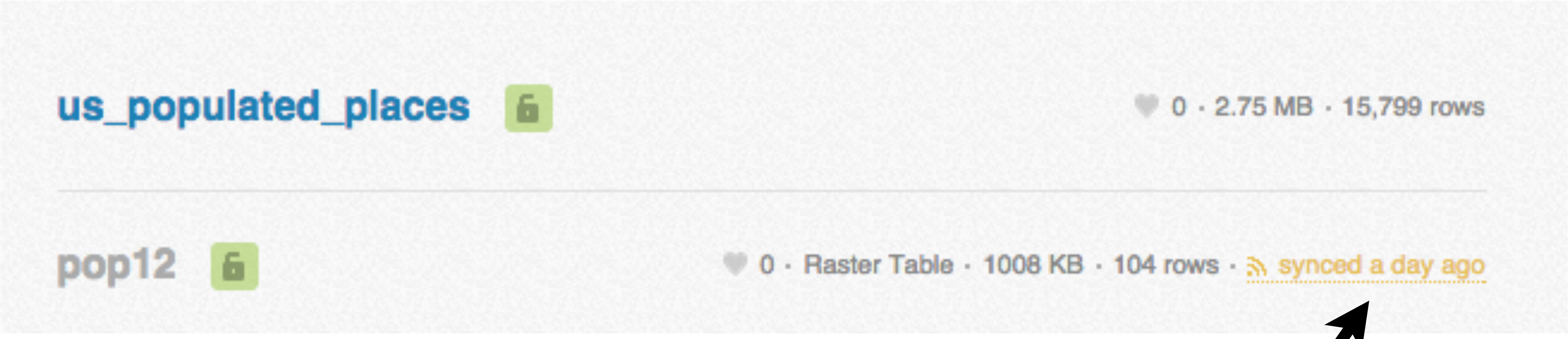
Upload to

amazon web services™ S3

In order to keep Carnac up to date,
I have a process running on my home computer every 6 hours,
that pulls the precipitation forecast from NOAA,
converts it into a GeoTIFF,
and then stuffs it up into an S3 bucket.

CARTODB

Geospatial on the cloud



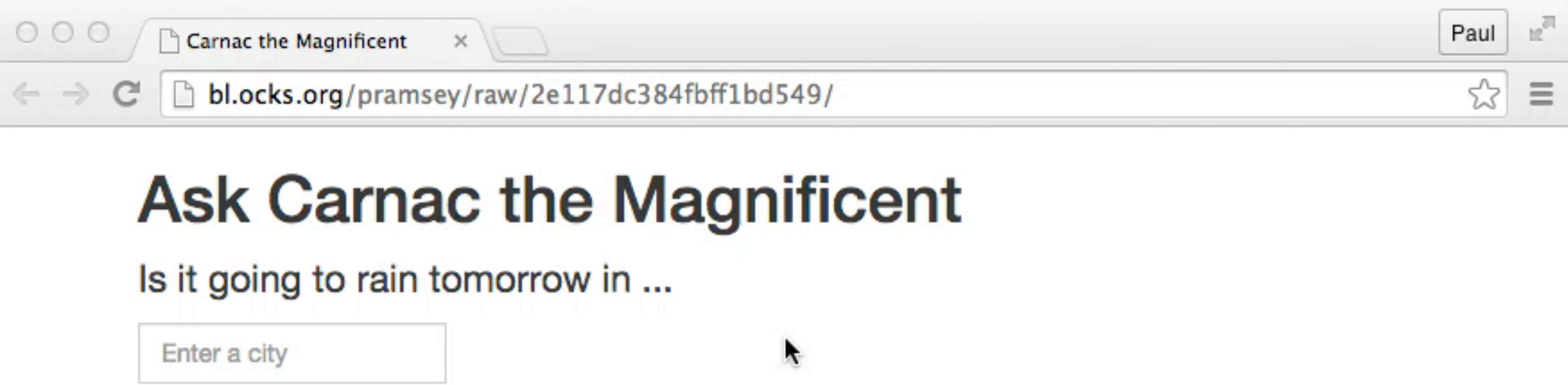
us_populated_places 🔓 ♥ 0 · 2.75 MB · 15,799 rows

pop12 🔓 ♥ 0 · Raster Table · 1008 KB · 104 rows · synced a day ago

Download From

amazon web services™ S3

From there, CartoDB automatically syncs it every day or so,
using the standard table sync capability of the platform.
(Basically, every refresh period,
it just slurps the whole file down and replaces the current data with the new data).



Ask Carnac the Magnificent

Is it going to rain tomorrow in ...

Demo http://bl.ocks.org/pramsey

And that’s Carnac the Magnificent, looking into the future.

“look into the future”

ok so that was the future, what about the past, can we do anything there?
how can we record history?

as the cynics
say, history is written
by the victors

well, taking the usual historical perspective, if we want to write history, first we have to win, we have to be in control, like a DBA. <X> and then we need some triggers

as the ~~cynics~~ DBAs
say, history is written
by the ~~victors~~ triggers

well, taking the usual historical perspective, if we want to write history, first we have to win, we have to be in control, like a DBA. <X> and then we need some triggers

```
CREATE TABLE addresses (
    gid SERIAL PRIMARY KEY,
    addr VARCHAR,
    geom Geometry(Point, 4326)
);
```

So, suppose we have a very simple base table
(but this approach works for complex tables too)
to record history, we make a second table that exactly mirrors it in structure

```
CREATE TABLE addresses_history (
    gid INTEGER,
    addr VARCHAR,
    geom Geometry(Point, 4326),
    created TIMESTAMP,
    created_by VARCHAR(32),
    deleted TIMESTAMP,
    deleted_by VARCHAR(32),
    hid SERIAL PRIMARY KEY
);
```

Except in addition to the main columns it has some metadata columns,
a creation timestamp and owner
a deletion timestamp and owner
and unique primary key of its own

- Features are live when “created” < time < “deleted”

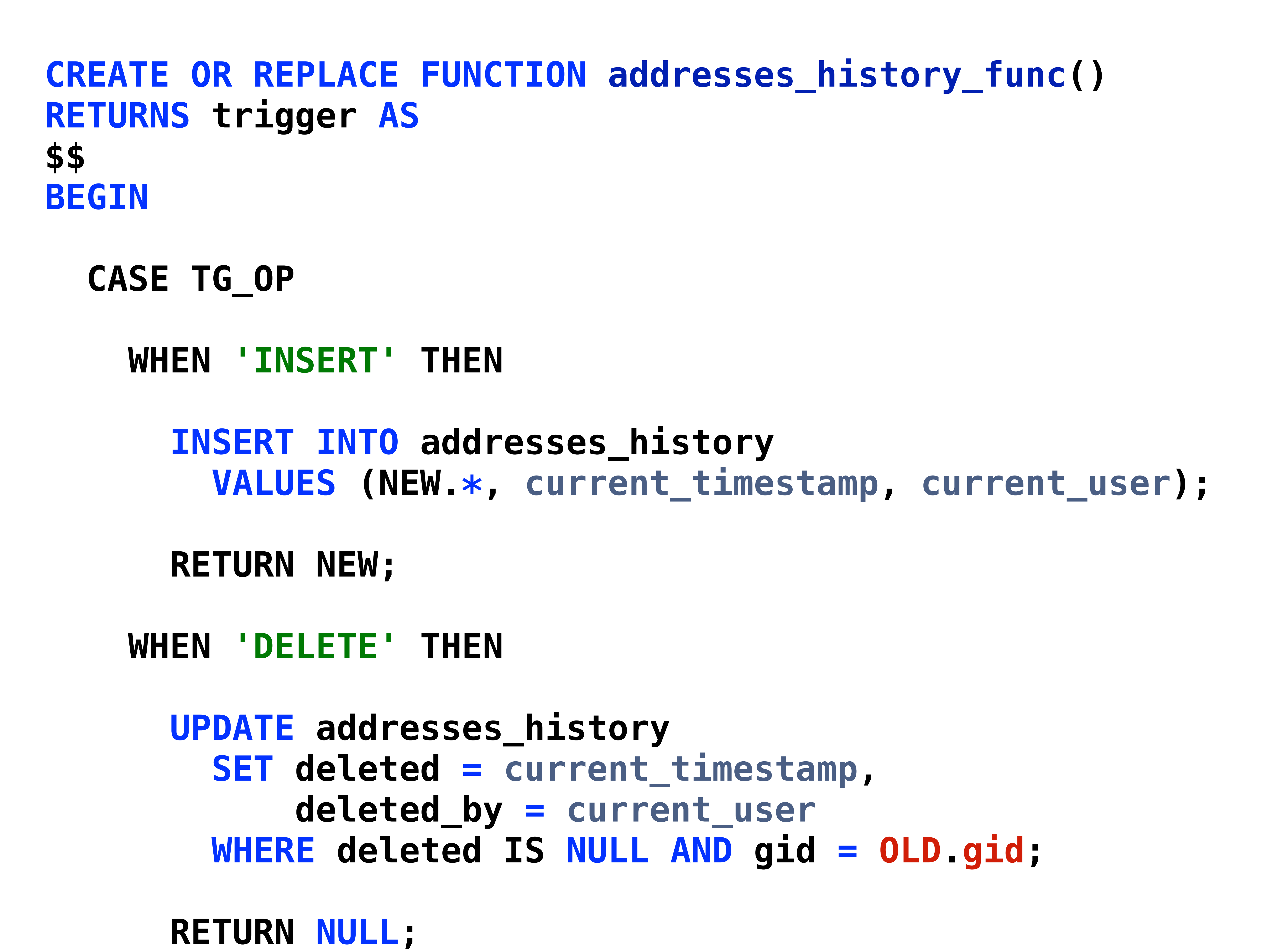
- Features are current when “deleted” IS NULL

- Current state on July 16:

```
SELECT * FROM addresses_history
WHERE created <= 'July 16, 2014'
AND ( deleted > 'July 16, 2014' OR
      deleted IS NULL )
```

Using this structure, we can figure out, for any given time, what features were “live”. That is what features had been created by that time, but had not yet been deleted.

So a query to find the state of data on July 16, would look like this,
- any record created before July 16,
- but not yet deleted on that date (or, not yet deleted at all)



```
CREATE OR REPLACE FUNCTION addresses_history_func()
RETURNS trigger AS
$$
BEGIN

  CASE TG_OP

    WHEN 'INSERT' THEN

      INSERT INTO addresses_history
        VALUES (NEW.*, current_timestamp, current_user);

      RETURN NEW;

    WHEN 'DELETE' THEN

      UPDATE addresses_history
        SET deleted = current_timestamp,
            deleted_by = current_user
        WHERE deleted IS NULL AND gid = OLD.gid;

      RETURN NULL;
```

In order to maintain the history log, we attach a trigger to the working table,
whenever a new record is inserted into the main table, the trigger inserts the same data into the history table, with the metadata about who inserted it, and when

```
CREATE OR REPLACE FUNCTION addresses_history_func()
RETURNS trigger AS
$$
BEGIN

  CASE TG_OP

    WHEN 'INSERT' THEN

      INSERT INTO addresses_history
        VALUES (NEW.*, current_timestamp, current_user);

      RETURN NEW;

    WHEN 'DELETE' THEN

      UPDATE addresses_history
        SET deleted = current_timestamp,
            deleted_by = current_user
        WHERE deleted IS NULL AND gid = OLD.gid;

      RETURN NULL;
```

In order to maintain the history log, we attach a trigger to the working table,
whenever a new record is inserted into the main table, the trigger inserts the same data into the history table, with the metadata about who inserted it, and when

```
  CASE TG_OP

    WHEN 'INSERT' THEN

      INSERT INTO addresses_history
        VALUES (NEW.*, current_timestamp, current_user);

      RETURN NEW;

    WHEN 'DELETE' THEN

      UPDATE addresses_history
        SET deleted = current_timestamp,
            deleted_by = current_user
        WHERE deleted IS NULL AND gid = OLD.gid;

      RETURN NULL;

    WHEN 'UPDATE' THEN

      UPDATE addresses_history
        SET deleted = current_timestamp,
            deleted_by = current_user
        WHERE deleted IS NULL and gid = OLD.gid;
```

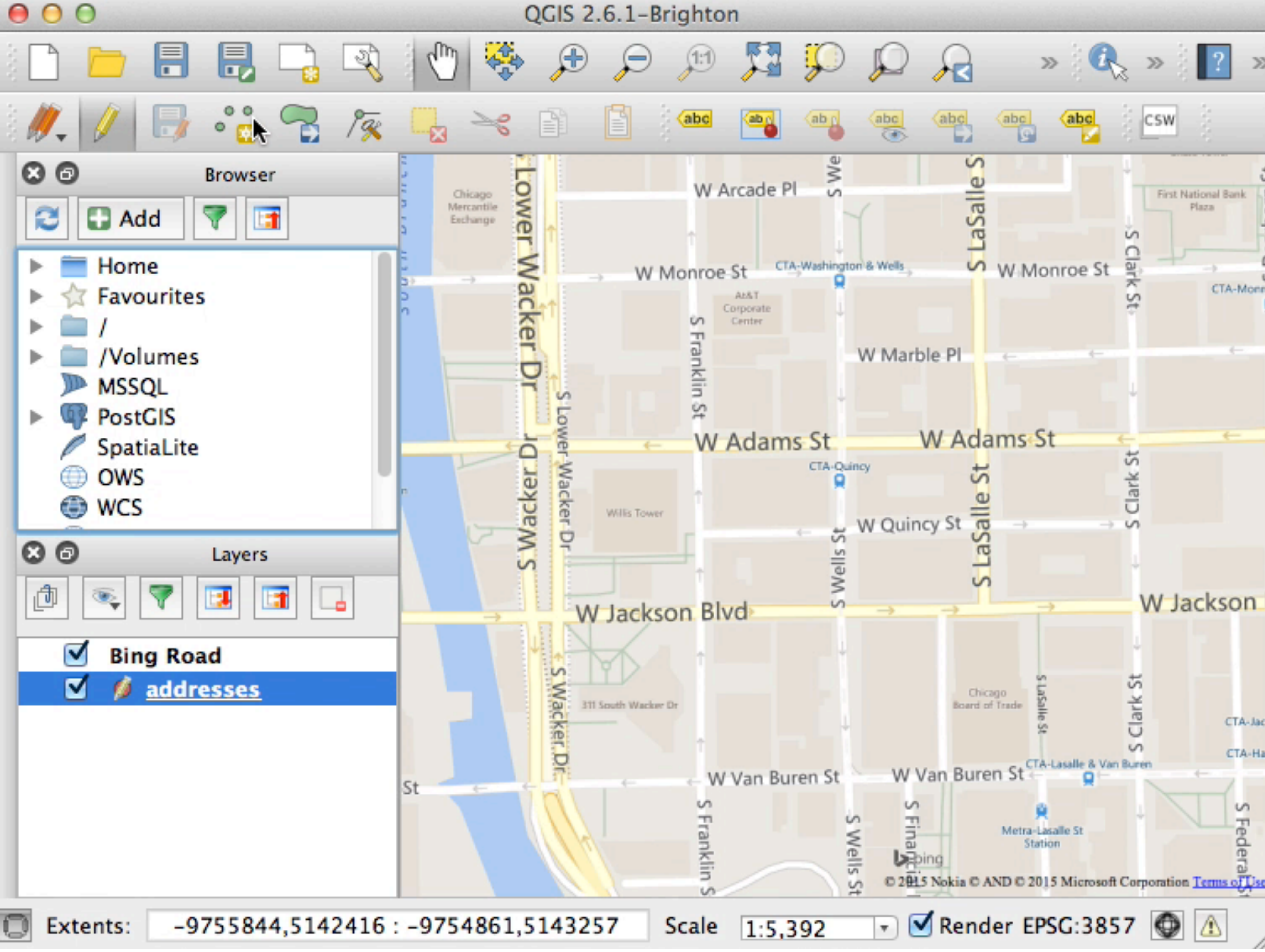
when ever a record is deleted, the trigger updates the history table, noting who did the deleting, and when the deletion occured

```
        SET deleted = current_timestamp,
            deleted_by = current_user
        WHERE deleted IS NULL AND gid = OLD.gid;

      RETURN NULL;

    WHEN 'UPDATE' THEN

      UPDATE addresses_history
        SET deleted = current_timestamp,
            deleted_by = current_user
        WHERE deleted IS NULL and gid = OLD.gid;

      INSERT INTO addresses_history
        VALUES (NEW.*, current_timestamp, current_user);

      RETURN NEW;

  END CASE;

END;
$$
LANGUAGE plpgsql;
```

finally, an update is handled as a deletion of the current state, followed by an insertion of the new state, at the current time.

```
CREATE TRIGGER
  addresses_history_trigger
AFTER
  INSERT OR
  UPDATE OR
  DELETE
ON addresses
  FOR EACH ROW
  EXECUTE PROCEDURE
    addresses_history_func();
```

With that function in place, we just tie it to the main table as a trigger on INSERT, UPDATE and DELETE, to keep the history table in sync.
And here's what it looks like in action.
Starting from an empty addresses table....



Extents: -9755844,5142416 : -9754861,5143257 Scale 1:5,392 ☑ Render EPSG:3857

I set up an actual database with the addresses and history tables and the trigger enabled, and then I added three records, using our favourite editor, QGIS.

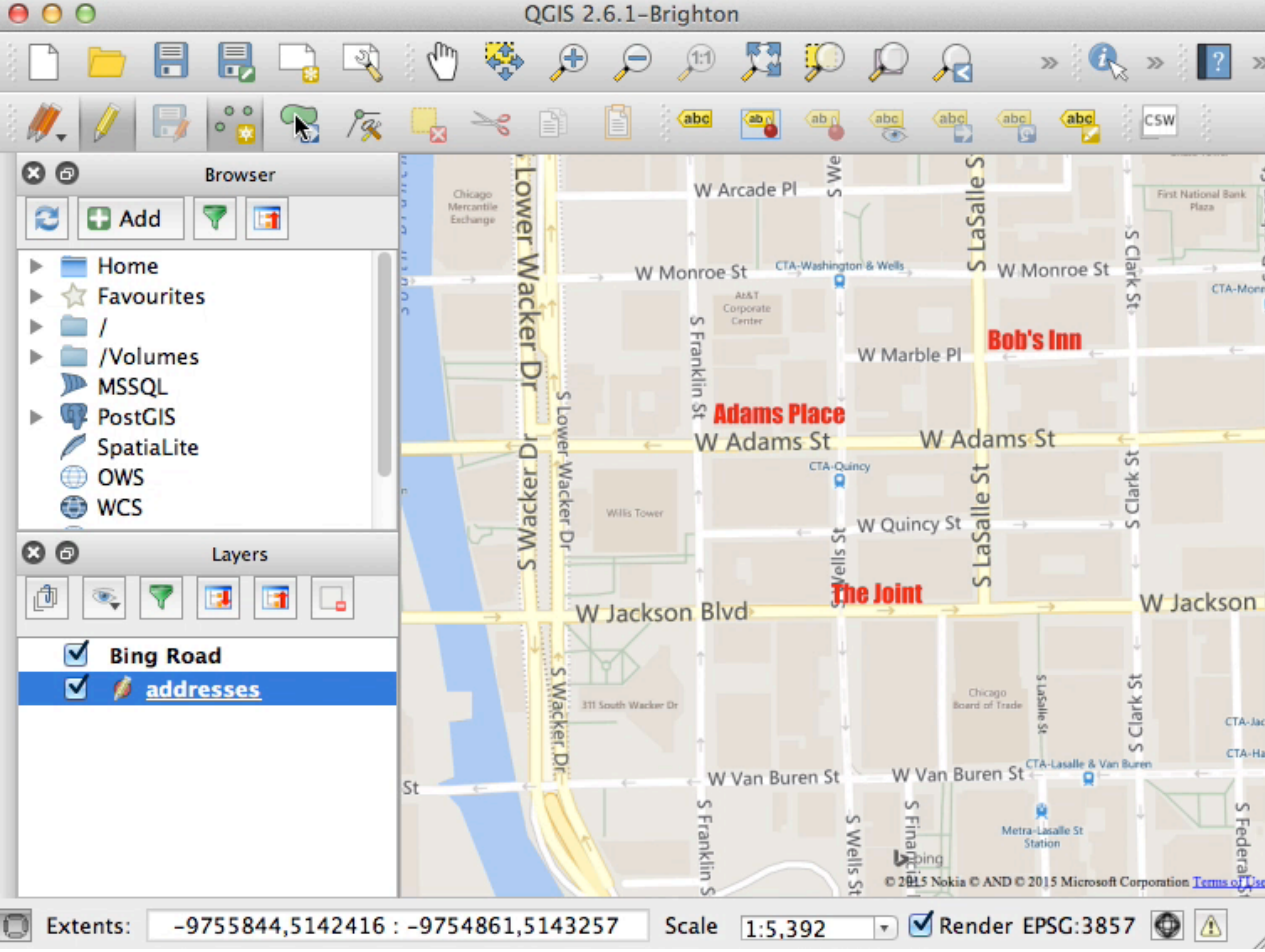
The part I like about this way of tracking history, doing it in the database, is that it doesn't matter how you generate the edits. With QGIS, with a web service, with a JDBC application, with hand-run bits of SQL on the command line.

No matter how you generate the edits, the history is saved the same way, it's *always* tracked. So this kind of history tracking is robust over time: the infrastructure around the system can change a lot while the history keeps on trucking.

addresses_history

```
 gid |    addr     |          created           |          deleted           | hid
-----+-------------+----------------------------+----------------------------+-----
   1 | Bob's Inn   | 2015-02-17 12:18:06.761291 |                            |   1
   2 | The Joint   | 2015-02-17 12:18:06.761291 |                            |   2
   3 | Adams Place | 2015-02-17 12:18:06.761291 |                            |   3
```

So, after performing those edits, I had three records in my main table, and *also* three records in my history table. The difference being that the history table records knew their creation date. Since they had not been deleted yet, the deleted field is still NULL.
You can also see when I made the screencast, from the creation dates.



Next, I used QGIS to edit the records,
changing the location of each one.
I hit save for each one separately, so they'd all have distinct change dates,
otherwise QGIS would just submit the changes in a batch and you couldn't tell which one I edited first.

addresses_history

```
 gid |    addr     |          created           |          deleted           | hid
-----+-------------+----------------------------+----------------------------+-----
   1 | Bob's Inn   | 2015-02-17 12:18:06.761291 |                            |   1
   2 | The Joint   | 2015-02-17 12:18:06.761291 |                            |   2
   3 | Adams Place | 2015-02-17 12:18:06.761291 |                            |   3
```

```
 gid |    addr     |          created           |          deleted           | hid
-----+-------------+----------------------------+----------------------------+-----
   1 | Bob's Inn   | 2015-02-17 12:18:06.761291 | 2015-02-17 12:18:23.56757  |   1
   2 | The Joint   | 2015-02-17 12:18:06.761291 | 2015-02-17 12:18:28.448761 |   2
   3 | Adams Place | 2015-02-17 12:18:06.761291 | 2015-02-17 12:18:17.064956 |   3
   3 | Adams Place | 2015-02-17 12:18:17.064956 |                            |   4
   1 | Bob's Inn   | 2015-02-17 12:18:23.56757  |                            |   5
   2 | The Joint   | 2015-02-17 12:18:28.448761 |                            |   6
```

Now the history table has three more records! The original insertions have been marked as deleted, and for each deletion there is a replacing addition record for the current state of the data.

addresses_history

```
 gid |    addr     |          created           |          deleted           | hid
-----+-------------+----------------------------+----------------------------+-----
   1 | Bob's Inn   | 2015-02-17 12:18:06.761291 |                            |   1
   2 | The Joint   | 2015-02-17 12:18:06.761291 |                            |   2
   3 | Adams Place | 2015-02-17 12:18:06.761291 |                            |   3
```

```
 gid |    addr     |          created           |          deleted           | hid
-----+-------------+----------------------------+----------------------------+-----
   1 | Bob's Inn   | 2015-02-17 12:18:06.761291 | 2015-02-17 12:18:23.56757  |   1
   2 | The Joint   | 2015-02-17 12:18:06.761291 | 2015-02-17 12:18:28.448761 |   2
   3 | Adams Place | 2015-02-17 12:18:06.761291 | 2015-02-17 12:18:17.064956 |   3
   3 | Adams Place | 2015-02-17 12:18:17.064956 |                            |   4
   1 | Bob's Inn   | 2015-02-17 12:18:23.56757  |                            |   5
   2 | The Joint   | 2015-02-17 12:18:28.448761 |                            |   6
```

Now the history table has three more records! The original insertions have been marked as deleted, and for each deletion there is a replacing addition record for the current state of the data.

addresses_history

```
 gid |    addr     |          created           |          deleted           | hid
-----+-------------+----------------------------+----------------------------+-----
   1 | Bob's Inn   | 2015-02-17 12:18:06.761291 |                            |   1
   2 | The Joint   | 2015-02-17 12:18:06.761291 |                            |   2
   3 | Adams Place | 2015-02-17 12:18:06.761291 |                            |   3
```

```
 gid |    addr     |          created           |          deleted           | hid
-----+-------------+----------------------------+----------------------------+-----
   1 | Bob's Inn   | 2015-02-17 12:18:06.761291 | 2015-02-17 12:18:23.56757  |   1
   2 | The Joint   | 2015-02-17 12:18:06.761291 | 2015-02-17 12:18:28.448761 |   2
   3 | Adams Place | 2015-02-17 12:18:06.761291 | 2015-02-17 12:18:17.064956 |   3
   3 | Adams Place | 2015-02-17 12:18:17.064956 |                            |   4
   1 | Bob's Inn   | 2015-02-17 12:18:23.56757  |                            |   5
   2 | The Joint   | 2015-02-17 12:18:28.448761 |                            |   6
```

Now the history table has three more records! The original insertions have been marked as deleted, and for each deletion there is a replacing addition record for the current state of the data.

addresses_history

```
SELECT * FROM addresses_history
WHERE created <= '2015-02-18'
AND (
  deleted > '2015-02-18'
  OR deleted IS NULL
);
```

If I want to see the past state of the data, for any data, I just plug a query into the history table,
asking for all records created before the date of interest, but not yet deleted on the date of interest.

addresses_history

```
SELECT *
FROM addresses_history
WHERE deleted_by = 'pramsey';
```

Or, if I want to go back and do an audit of the changes made
by a PARTICULAR user, it’s easy to pull just those changes out of
the history table too.
Things like reverting the changes of a certain user or changes associated
with a certain edit session get much easier when you maintain a full edit log on your data.

text search
federation
time

Thursday, March 12, 15

So, we’ve talking a bit about full-text search handling, and federated systems, and handling time a bit, but actually there’s so many more PostgreSQL features we could talk about, because

your database is beautiful

postgresql is such a beautiful, beautiful database,
so here’s your homework
when you get back home,
open up the PostgreSQL and learn and experiment a bit with

- Time ranges

```
SELECT '2011-04-07'::timestamp -
       '2010-09-04'::timestamp;
```

- Arrays

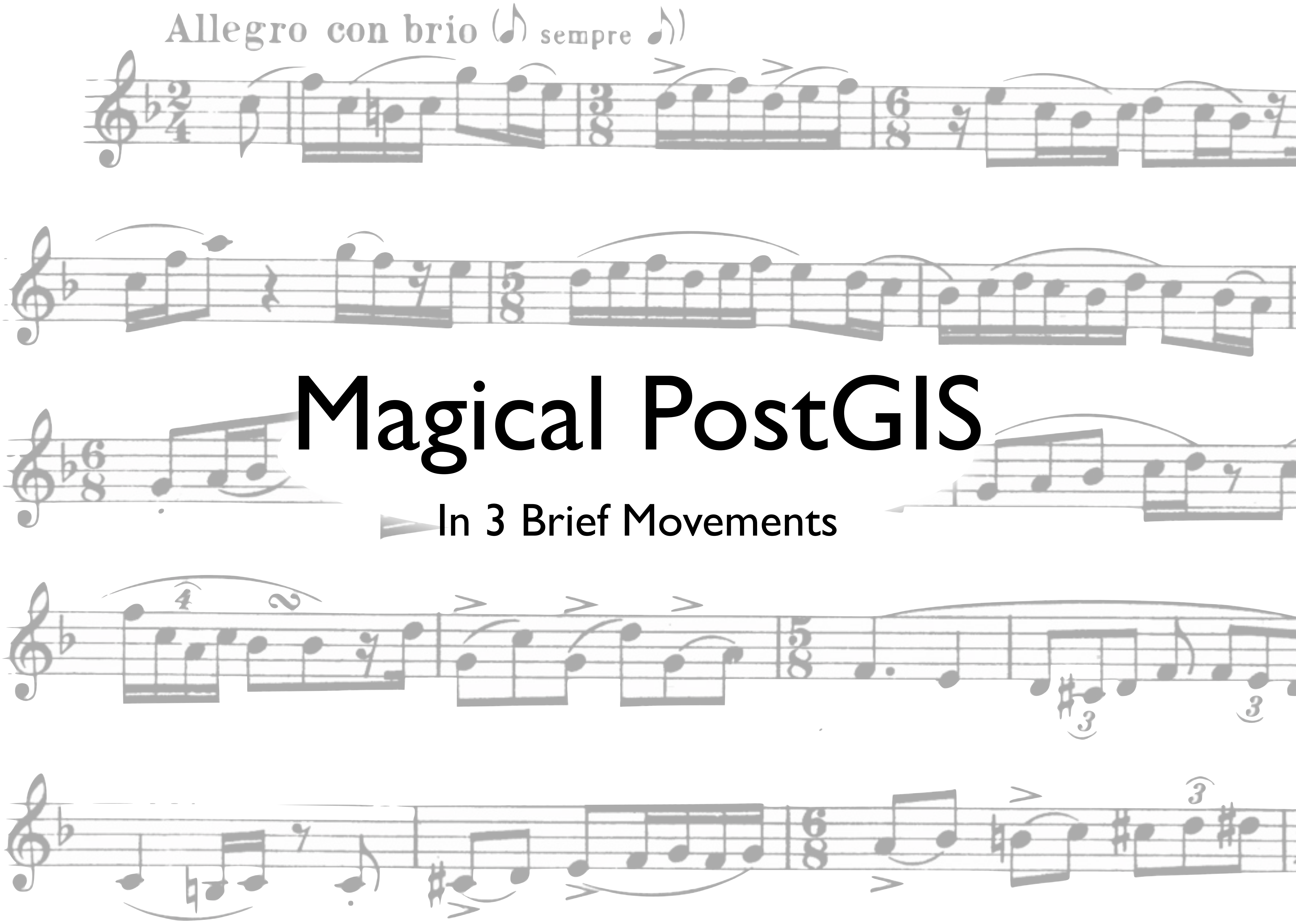
```
SELECT ARRAY[1,2,3] @> ARRAY[2];
```

- Regular Expressions

```
SELECT 'caats' ~ 'c.*ts';
```

time ranges, and the support for time in PostgreSQL in general
arrays, especially things like array operators and functions and aggregates
regular expressions, not just the operators, which are good for search, but also the functions, that can be used for advanced data editing and cleansing

all these features are extremely well implemented in postgresql
but poorly implemented in other database, if done at all

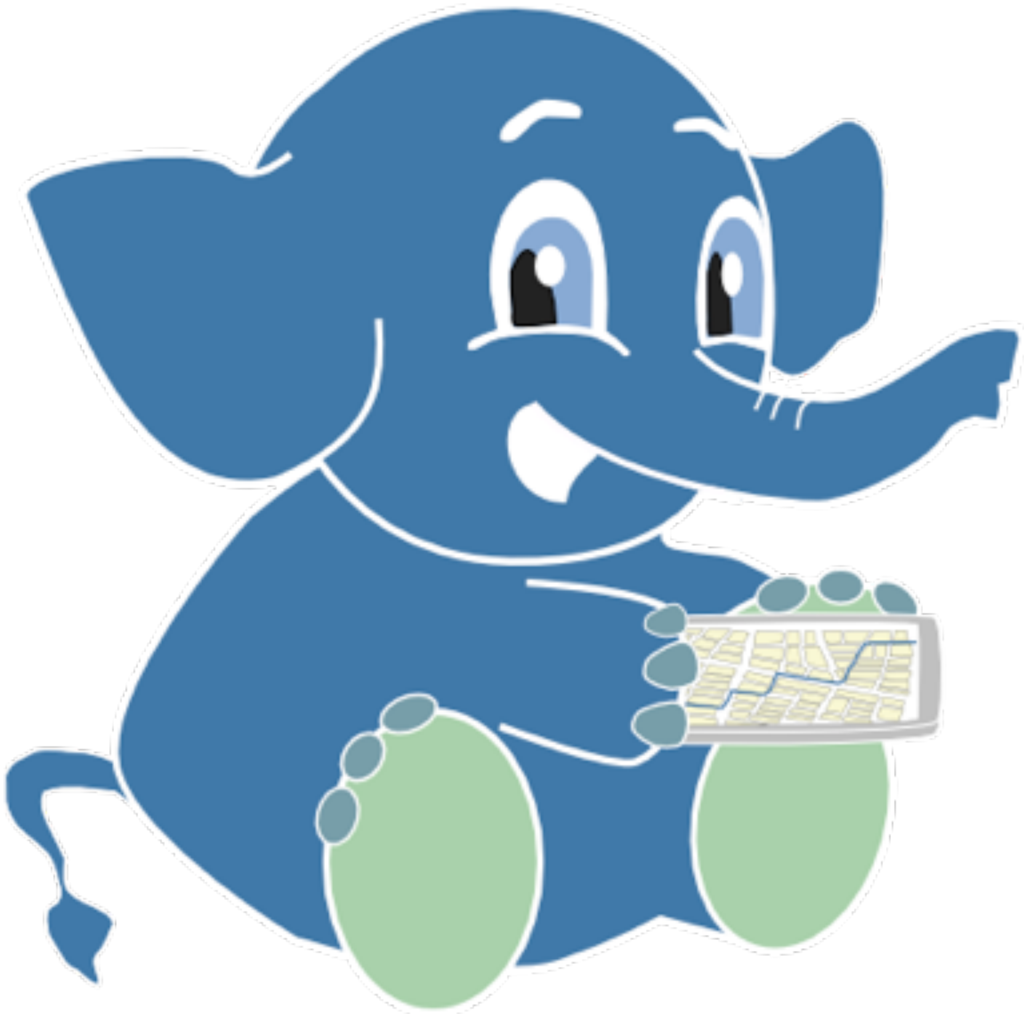


Magical PostGIS

In 3 Brief Movements

that’s been Magic PostGIS in three brief movements, thank you!

pgRouting: A Practical Guide



http://locate PRESS.com/pgrouting

Open Source "Geo" Books & Training