Spatial

Serving Dynamic Vector Tiles from PostGIS

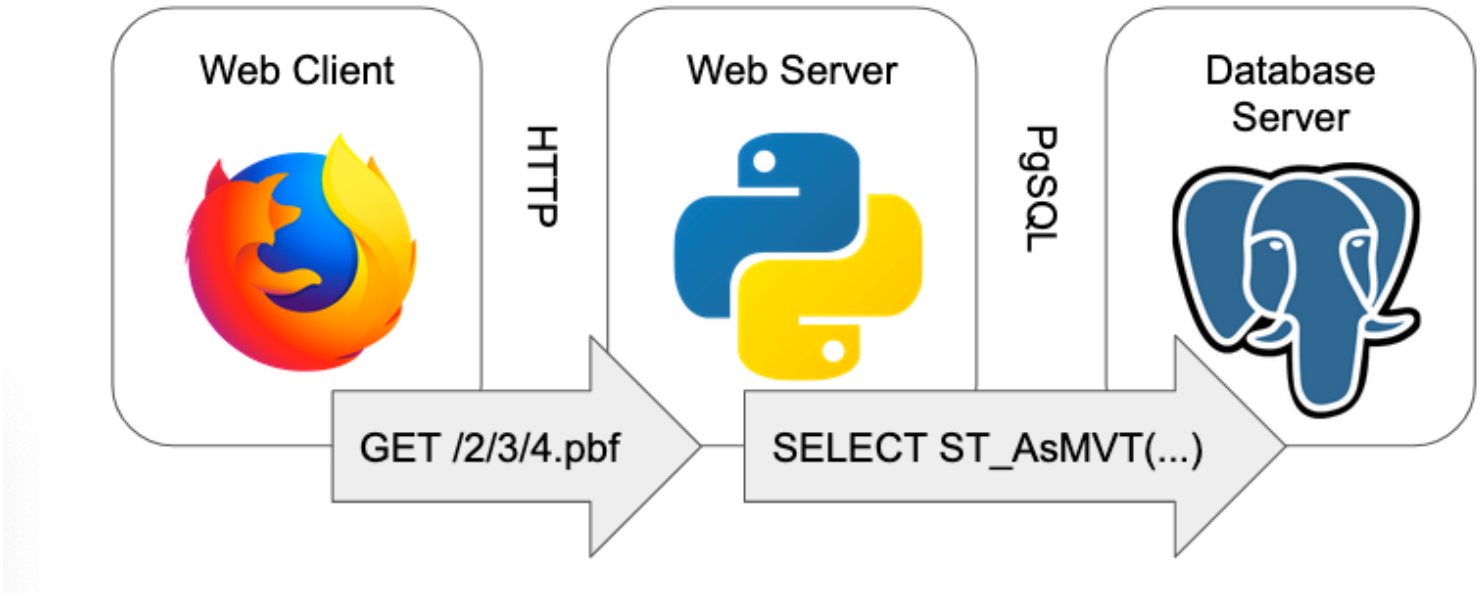
Paul Ramsey
Jul 9, 2019 · 6 min read

One of the most popular features of PostGIS 2.5 was the introduction of the "vector tile" output format, via the ST_AsMVT() function.

Vector tiles are a transport format for efficiently sending map data from a server to a client for rendering. The vector tile specification describes how raw data are quantized to a grid and then compressed using delta-encoding to make a very small package.

Prior to ST_AsMVT(), if you wanted to produce vector tiles from PostGIS you would use a rendering program (MapServer, GeoServer, or Mapnik) to read the raw data from the database, and process it into tiles.

With ST_AsMVT() it is now possible to move all that processing into the database, which opens up the possibility for very lightweight tile services that do little more than convert map tile requests into SQL for the database engine to execute.



There are already several examples of such light-weight services.

- Dirt-Simple PostGIS HTTP API
- Postile
- Martin

However, for learning purposes, here's a short example that builds up a tile server and map client from scratch.

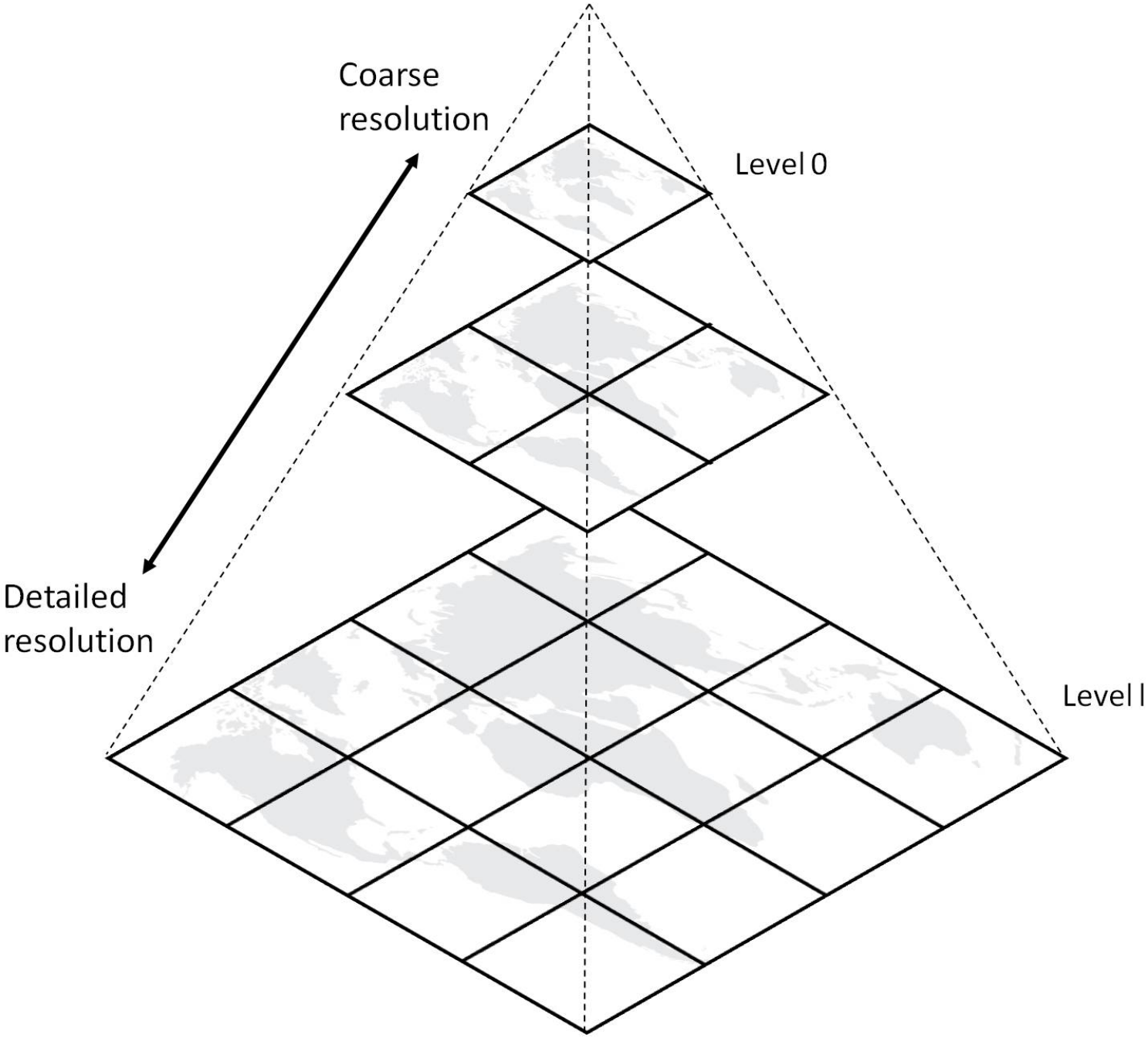
- Minimal MVT Server

This minimal tile server is in Python, but there's no reason you couldn't execute one in any language you like: it just has to be able to connect to PostgreSQL, and run as an HTTP service.

What are Tiles

A digital map is theoretically capable of viewing data at any scale, and for any region of interest. Map tiling is a way of constraining the digital mapping problem, just a little, to vastly increase the speed and efficiency of map display.

- Instead of supporting any scale, a tiled map only provides a limited collection of scales, where each scale is a factor of two more detailed than the previous one.
- Instead of rendering data for any region of interest, a tiled map only renders it over a fixed grid within the scale, and composes arbitrary regions by displaying appropriate collections of tiles.

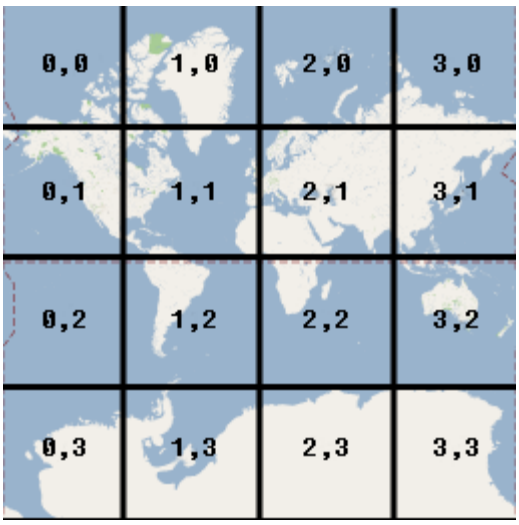


Most tile maps divide the world by starting with a single tile that encompasses the entire world, and calls that "zoom level 0". From there, each succeeding "zoom level" increases the number of tiles by a factor of 4 (twice as many vertically and twice and many horizontally).

Tile Coordinates

Any tile in a tiled map can be addressed by referencing the zoom level it is on, and its position horizontally and vertically in the tile grid. The commonly used "XYZ" addressing scheme counts from zero, with the origin at the top left.

This example is for zoom level 2 (`2^zoom = 4` tiles per size).



The web addresses of tiles in the "XYZ" scheme embed the "zoom", "x" and "y" coordinates into a web address:

```
http://server/{z}/{x}/{y}.format
```

For example, you can see the tile that encompasses Australia (zoom=2, x=3, y=2) in the tilesets of a number of map providers:

- https://tile.openstreetmap.org/2/3/2.png
- http://a.basemaps.cartocdn.com/light_all/2/3/2.png
- http://a.tile.stamen.com/toner/2/3/2.png

Building a map tile in the database involves feeding data through not one, but two PostGIS functions:

- ST_AsMVTGeom()
- ST_AsMVT()

Vector features in a MVT map tile are highly processed, and the ST_AsMVTGeom() function performs that processing:

- clip the features to the tile boundary;
- translate from cartesian coordinates (relative to geography) to image coordinates (relative to top left of image);
- remove extra vertices that will not be visible at tile resolution; and,
- quantize coordinates from double precision to the tile resolution in integers.

So any query to generate MVT tiles will involve a call to ST_AsMVTGeom() to condition the data first, something like:

```
SELECT ST_AsMVTGeom(geom) AS geom, column1, column2
FROM myTable
```

The MVT format can encode both geometry and attribute information, in fact that is one of the things that makes it so useful: client-side interactions can be much richer when both attributes and shapes are available on the client.

In order to create tiles with geometry and attributes, ST_AsMVT() function takes in a record type. So SQL calls that create tiles end up looking like this:

```
  SELECT ST_AsMVTGeom(geom) AS geom, column1, column2
  FROM myTable
) mvtgeom
```

We'll see this pattern again as we build out the SQL queries generated by the tile server.

Tile Server

The job of our minimal web tile server is to convert from tile coordinates, to a SQL query that creates an equivalent vector tile.

First the pathToTile function strips out the x, y and z components from the request.

Then tileIsValid confirms that the values make sense. Each zoom level can only have tile coordinates between `0` and `2^zoom - 1` so we check that values are in range.

"XYZ" tile maps are usually in a projection called "spherical mercator" that has the nice property of forming a neat square, about 40M meters on a side, over (most of) the earth at zoom level zero.

From the that square starting point, tileToEnvelope subdivides it to find the size of a tile at the requested zoom level, and then the coordinates of the tile in the mercator projection.

Now we can start constructing the SQL to generate the MVT format tile. First envelopeToBoundsSQL converts our envelope in python into SQL that will generate an equivalent envelope in the database we can use to query and clip the raw data.

With the bounds SQL we are now ready to calculate the full MVT-generating SQL statement in envelopeToSQL:

```
WITH
bounds AS (
    SELECT {env} AS geom,
           {env}::box2d AS b2d
),
mvtgeom AS (
    SELECT ST_AsMVTGeom(ST_Transform(t.{geomColumn}, 3857), bounds.b2d) AS geo
           {attrColumns}
    FROM {table} t, bounds
    WHERE ST_Intersects(t.{geomColumn}, ST_Transform(bounds.geom, {srid}))
)
SELECT ST_AsMVT(mvtgeom.*) FROM mvtgeom
```

And finally run the SQL against the database in sqlToPbf and return the MVT as a byte array.

That's it! The main HTTP do_GET callback for the script just runs those functions in order and sends the result back.

```
# Handle HTTP GET requests
def do_GET(self):

    tile = self.pathToTile(self.path)
    if not (tile and self.tileIsValid(tile)):
        self.send_error(400, "invalid tile path: %s" % (self.path))
        return
```

```
    self.log_message("path: %s\ntile: %s\n env: %s" % (self.path, tile, env))
    self.log_message("sql: %s" % (sql))

    self.send_response(200)
    self.send_header("Access-Control-Allow-Origin", "*")
    self.send_header("Content-type", "application/vnd.mapbox-vector-tile")
    self.end_headers()
    self.wfile.write(pbf)
```

Now we have a python client we can run that will convert HTTP tile requests into MVT-tile responses directly from the database.

```
http://localhost:8080/4/3/4.mvt
```

Map Client

Now that tiles are published, we can add our live tile layer to any web map that supports MVT format. Two of the most popular are

- OpenLayers, and
- Mapbox GL JS.

Map clients convert the state of a map windows into HTTP requests for tiles to fill up a map window. If you've used a modern web map, like Google Maps, you've used a standard web map -- they all work the same way.

OpenLayers

The OpenLayers map client has been built out using the NPM module system, and can be installed into an NPM development environment as easily as:

```
npm install ol
```

The example OpenLayers map for this post combines a standard raster base layer with an active layer from a PostgreSQL database accessed via our tile server.

```
var vtLayer = new VectorTileLayer({
  declutter: false,
  source: new VectorTileSource({
   format: new MVT(),
   url: 'http://localhost:8080/{z}/{x}/{y}.pbf'
  }),
  style: new Style({
      stroke: new Stroke({
         color: 'red',
         width: 1
        })
    })
});
```

Mapbox GL JS

The Mapbox GL JS is more tightly bound to the Mapbox ecosystem, but can be run without using Mapbox services or a Mapbox API key.

understanding the style language that is used to specify both map composition and the styling of vector data in the map.

WRITTEN BY

Paul Ramsey

July 9, 2019 • More by this author

CRUNCHY DATA NEWSLETTER

Subscribe to the Crunchy Data Newsletter to receive Postgres content every month.

Enter your email

Join The List