Pie Insurance 2 Lead Generation Heatmap

Team

Sean Keenan

Nicholas Stevenson

Lucas Bowar

Pouya Zakeri

Clients

Regan Henry

Mark Rogers

6/13/2022

PIE INSURANCE

Introduction

Pie Insurance is an insurance company focused on small business workers compensation. They have offices based in Denver and Washington DC. Pie Insurance offers policies to businesses, which outlines what is covered by claims. These policies also include data such as business name, location, size of business, and hazard group. Hazard groups, labeled A-F, detail how likely a business is to file a claim. For example, a construction company will be more likely to file a claim than an accounting firm, and will thus belong to a different hazard group.

The Lead Generation Heatmap project is an internal visual tool displaying the policies on a map of the continental United States. Pie currently offers over 26,000 policies across the United States, and as the company grows, they want a representation of their impact. This project is meant to help employees better visualize where the policies Pie offers are located in order to make business decisions, demonstrate impact to future clients, or to display in the office as a memoir of how far the company has come.

Requirements

Functional Requirements:

- Using H3 tiles to display data
  - H3 Tiles referenced in the Appendix
  - Written in Node.js
  - Using React framework
  - Color coded based on density of policies in each tile
- When H3 tiles are clicked, a detail sidebar should display
  - Detailed sidebar should give more info about the specific tile, including number of policies, location, etc.
- Zoom functionality is included: user is able to increase or decrease resolution
- Policy data read from Snowflake should be written to an internal AWS DynamoDB database
- Policy data is aggregated into H3 tile data
  - Variety of tile resolutions included to allow for zoom functionality
- Process is automated
  - Querying data from Snowflake, aggregation, display, etc.
  - Allows for the map to be updated with new data regularly
- Safe and efficient movement of data (see section titled System Architecture)
  - Client data must be secure

Non-functional Requirements:

- End product should be scalable
    - Any goals not reached should be easy to implement, given our project
- Output should be aesthetically pleasing
- Output should be easy to understand, and give insight to those viewing it
- Path of least permissions should be taken
    - Data should only be accessed that is directly used in the map
    - Permissions should only be granted if they are used in the project
- Cost of the system should be considered: namely runtime and storage used in AWS

System Architecture

The data pipeline is demonstrated graphically in Figure 1. Detailed descriptions of architecture components described below.

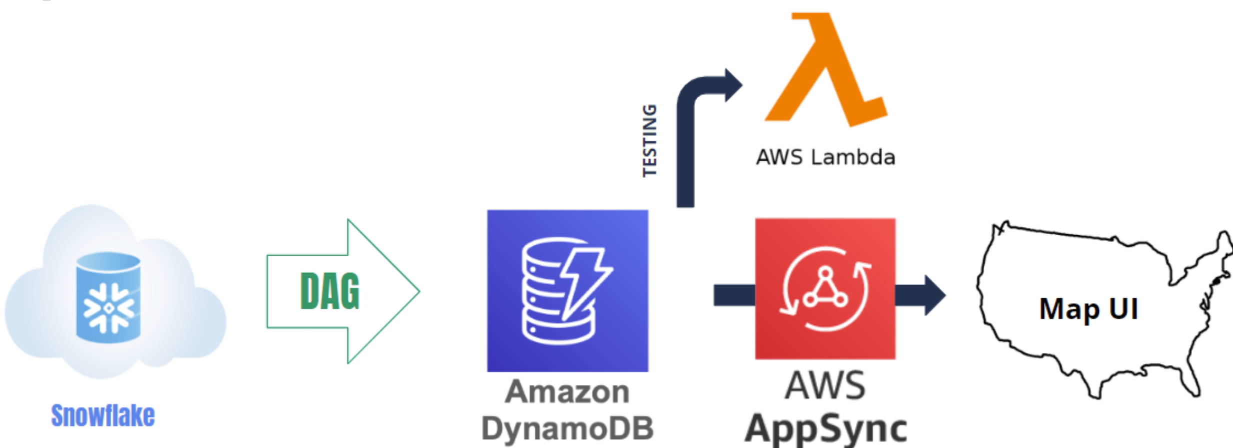

Figure 1: System Architecture

The policy data was initially stored in Snowflake, an external cloud database. A collection of scripts written in Python and SQL were used to read the data from Snowflake and write to DynamoDB. A directed acyclic graph (DAG) organized these scripts to run one after another. This ensured the correct order of scripts and smoothly read the data from Snowflake and wrote it to DynamoDB. In these scripts, the policies are aggregated into H3 tiles using Python.

DynamoDB is a NoSQL, AWS cloud database. The DynamoDB table, labeled H3Tiles_feature02, was created using a CloudFormation template. CloudFormation is an AWS service that was leveraged to create the DynamoDB table automatically. In this template, the primary key of the table was declared to be the hash string of the H3 tile. Resolution, an attribute defined in the schema that determines the level of detail of each H3 tile, was defined as a global secondary index. This allowed the queries to filter by resolution.

Figure 2 demonstrates how data is transferred from the DynamoDB table to the frontend.

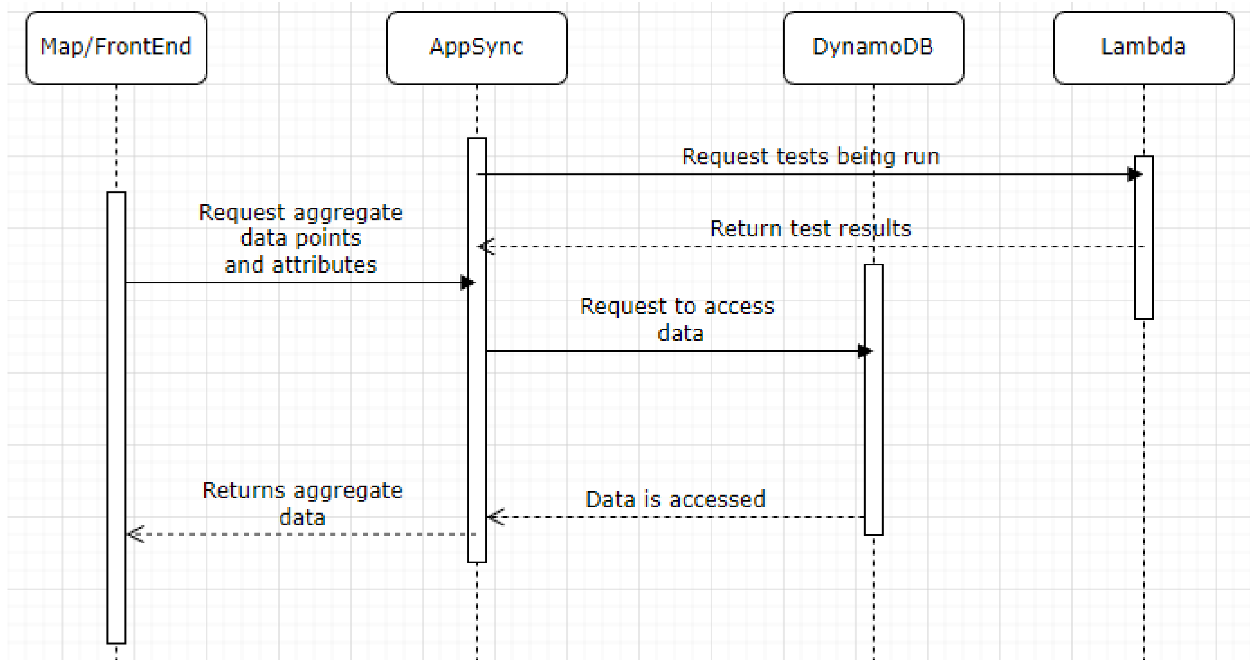

Figure 2: Sequence Diagram

The query language used was GraphQL. A H3Tile type was defined, with attributes tile hash, resolution, latitudes, longitudes, hazard groups, hazard group counts, center latitude, and center longitude. Two queries were also declared in GraphQL: one to get all H3 tiles, and one to get all H3 tiles that are of a resolution parameter. These queries were defined in AppSync. AppSync resolvers use mapping templates to determine what information the queries declared return. The predominant query used in the project is `getH3TileByResolution`, with a resolution parameter based on the zoom level of the map.

AWS Lambda was used for testing. Lambda functions are a method of running code against a database. In this project, the lambda functions query the AppSync resolvers, and compare the results against test cases. Dummy tiles were placed in the DynamoDB table specifically for testing. These tests were run automatically using github workflows. When a pull request is made against the main branch, the github workflows run the tests and prevent merging unless the tests pass.

Figure 3 represents a wireframe of the proposed design for the project.

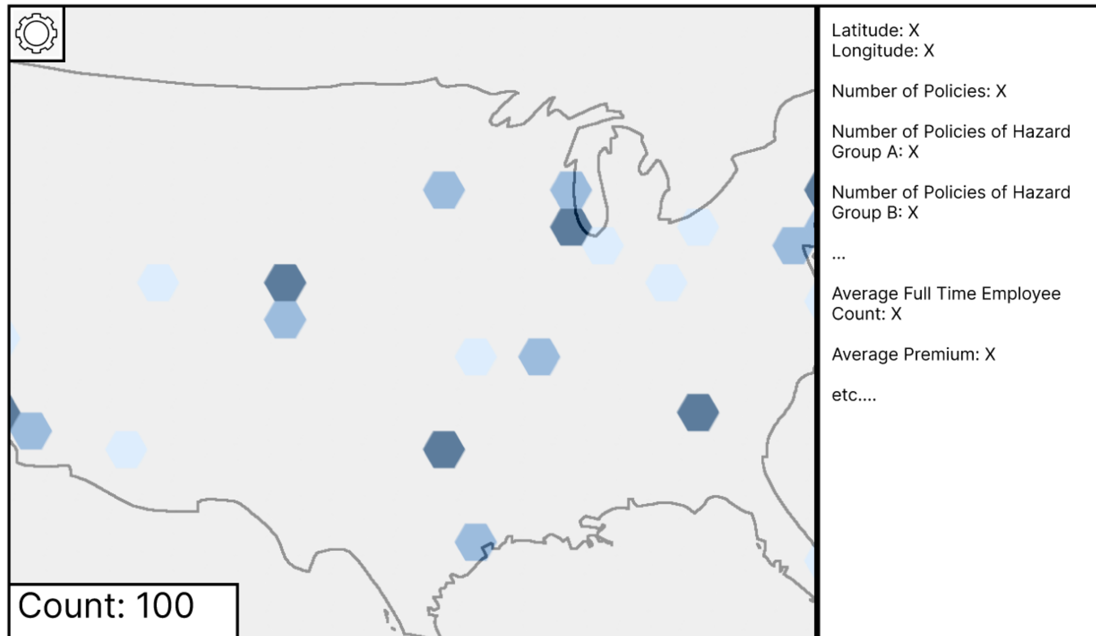

Figure 3: Wireframe of web page design

The front end queried AppSync resolvers for the resolution level determined by how zoomed in the map is. The more zoomed in the map, the higher the resolution. The data was all queried at the same time, and stored in front end data structures. The tiles were then plotted on the map with the tiles containing more policies being shaded darker. When a tile was clicked, detailed information about the tile was shown, including the number of policies.

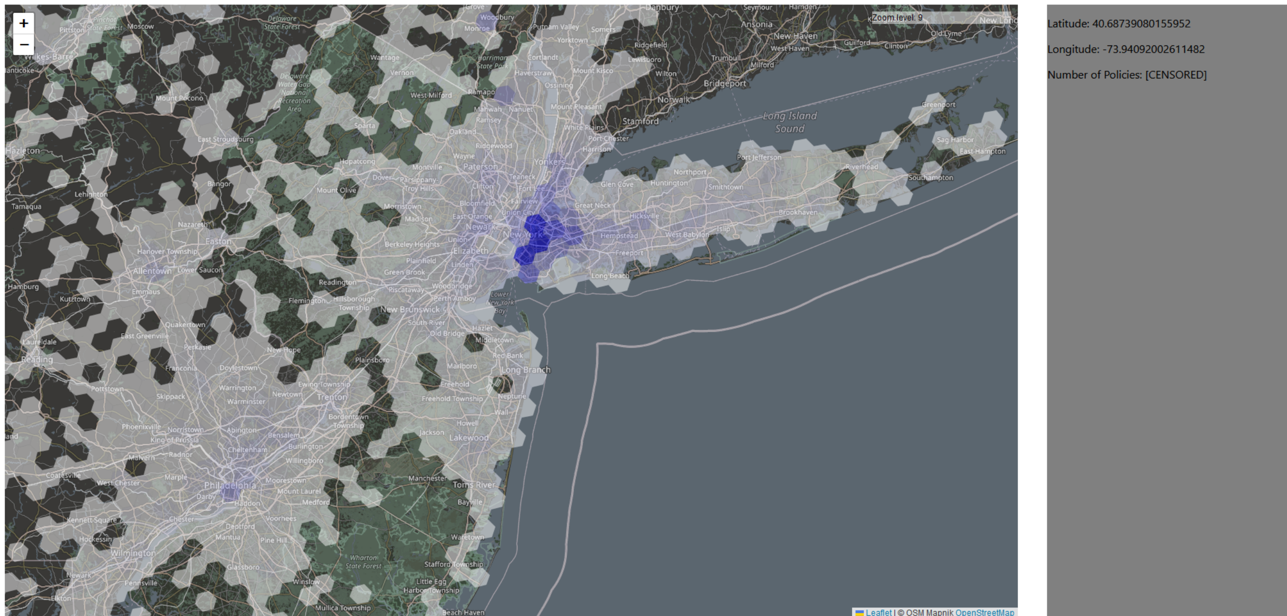

Figure 4: A screenshot of the heatmap

Technical Design

A GraphQL schema defines the structure of the table in DynamoDB and the queries that are resolved by AppSync. It is a major connection point between the front end and the back end, and solidifying the schema was crucial to our project. The schema is shown in Figure 5.

```
# DataVisualizationMap GraphQL Schema
schema {
  query: Query
}

# Returns list of all h3Tiles
type Query {
  getH3TileByResolution(resolution: Int!): [H3Tile!]!
  getAllH3Tiles: [H3Tile!]!
}

# Individual H3 tiles with tile hash for labeling the hexagon, number of policies,
# and the count of each hazard group. Coordinates are used to draw the hexagon.
type H3Tile {
  tileHash: String!
  numPolicies: Int
  hazardGroups: [String]
  hazardGroupCounts: [Int]
  lats: [Float]
  longs: [Float]
  centerLat: Float
  centerLong: Float
  resolution: Int
}
```

Figure 5: GraphQL schema

The H3 tile has several critical attributes. `tileHash` is used as our primary / partition key because it is guaranteed to be unique. `numPolicies` is the number of policies contained within the H3 tile, used to determine the color of the tile when displayed on the map. `lats` and `longs` are a set of coordinates used to plot the H3 tile. `resolution` determines what level of zoom is required on the map in order to see the tile.

The project required two queries on the DynamoDB database. The first, `getAllH3Tiles`, was to grab every single tile. This query was mainly used for testing, as the front end did not require every tile at the same time. The second query, `getH3TileByResolution`, was used to grab all H3 tiles of a certain resolution. This was used by the front end to grab only the appropriate tiles for the current zoom level on the map.

Another important part of our final design was how we implemented displaying the H3 tiles themselves. The first step to display them was to have the front end call the query itself to get the tiles from the DynamoDB table at a certain resolution. This resolution was determined based off of the current zoom level of the map. This step would return the H3 tiles in a json format which could be parsed.

Once the H3 tiles were sent to the front end, the front end could then parse through each H3 tile found in the json file. For each tile, the latitude and longitude were parsed and then stored as an array of points, where each point contained a latitude and a longitude. This array of points was then used to make clickable polygons. Some example data made into these polygons is shown on the map below in Figure 6.

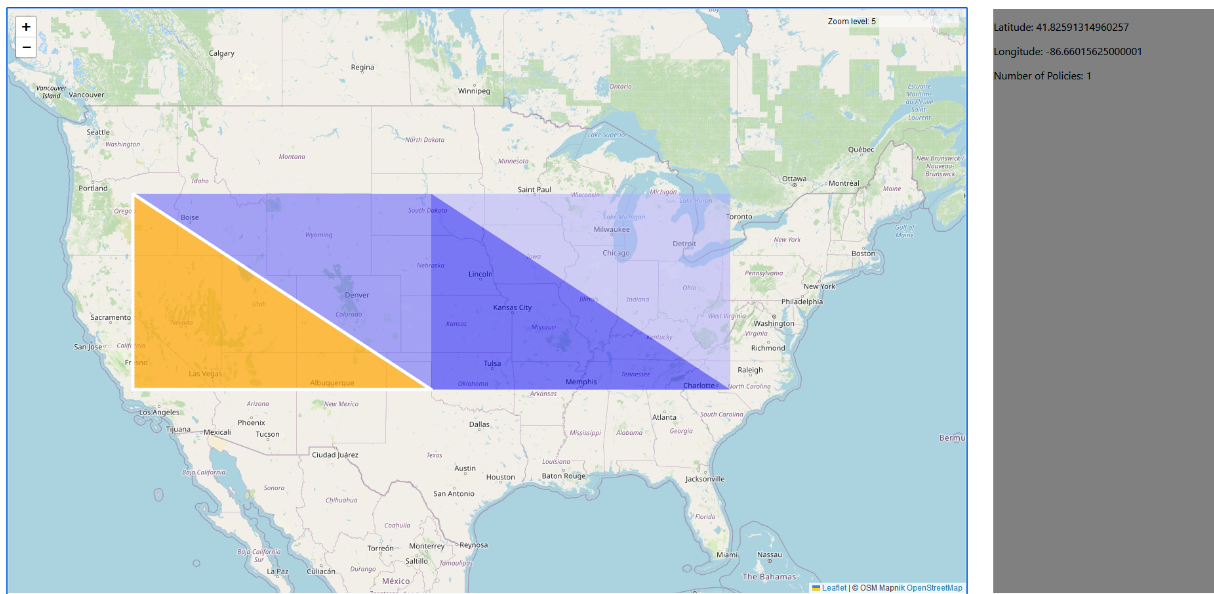

Figure 6: Example tiles

To actually make the polygons on the map, a library for javascript called Leaflet was used. Leaflet provided the tools for actually drawing the polygon on a map provided by Open Street Maps as well as setting up observers to perform actions when the polygon is interacted with. For instance an on click function was made to display the center latitude, center longitude and number of policies contained to the sidebar for each polygon.

QA

As our project would be used by Pie Insurance employees, it was important that we ensure that our project was of high quality.

One of the ways we ensured quality was by making unit testing and/or integration testing required on all of our tickets. This made sure that those tickets performed as expected for the most common use cases. In this context, a “ticket” refers to a chunk of work defined for the current sprint. In addition, Github integration testing was done through actions and workflows to make sure that branches merged into main passed required tests. Testing was done before tickets were merged into main to make sure that main only contained quality software. This was necessary to adhere with ACM principles 1.03 and 3.10.

Another way we ensured quality was by requiring that every pull request must have been reviewed by another person. The main branch was protected from merging unless that merge had been reviewed. As we had Github Actions configured this was absolutely required. This made sure that lower quality work or broken systems didn’t get added to the main branch of our project. This was necessary to adhere with ACM principle 3.01.

One other way we ensured quality was by meeting with our client regularly during our working process. This was integral as if we did anything incorrectly, she could catch our errors as she was more qualified than us. This manifested itself through frequent standups and frequent meetings for help on unfamiliar languages and concepts. This helped us adhere to ACM principle 3.04.

One final way we ensured quality on this project was by using Github Workflow for deployment. This removed possible human error in the steps to deploy our project, as the process became automated.

Results

The goal of this project was to create a heatmap displaying the density of Pie Insurance’s policies across the U.S. For testing, we had unit testing set up in the DAG written in Python, as well as in the DynamoDB instance and AppSync, through Lambda. The front end was exclusively tested manually, as the aesthetic of the front end is a non-functional requirement.

We successfully implemented the map with H3 tiles, as well as a basic informational sidebar that showed the latitude and longitude center of the tile, as well as the policy count within it. However, the sidebar wasn’t a fully achieved feature, as we had hoped to show additional information about the policies. For instance we had planned on displaying the number of policies

within that tile that belonged to a certain hazard group but that never made it into the final product. Additionally, we didn't reach the stretch goal of allowing the user to query the policies within the map. For example, looking at only policies from the last 3 years, or only policies from companies of a certain size. On the bright side, this goal as well as other modifications will be straightforward to implement, given the foundation.

Future Work

One of the ways our project could be improved upon in the future would be to add filters to the policy data that was displayed. The policy data stored in the Snowflake database came with many features that this data could be grouped by. For instance, each policy came with a hazard group it belonged to, which determined how likely Pie Insurance thinks a claim will be filed as part of that policy. A filter could be added so that only policies that belonged to certain hazard group(s) would be displayed. These filters could also be applied to various other features such as the premium or number of full time employees. This would allow a user to visualize more specific data from the heatmap. However it would require setting up a settings UI in the front end and more queries in the back end.

Another way our project could be modified/improved upon in the future would be to display the actual policies as points on a map if the user zooms in enough. As of writing this report our project only displayed a heat map of hex tiles which contained a number of policies. If this change is made, when the user zooms in beyond a certain predefined point, this heatmap of hex tiles would disappear and instead each policy itself would appear as a marker according to the exact location of the business holding that policy. This would allow a user to get a more granular view of the policies if he or she so desired. However it would require us making huge changes to the back end in order to allow us to query both hex tiles and the policies themselves.

One component that could be swapped on our project would be the hex tiles themselves. Instead of displaying a heat map of hex tiles, the map could instead display a heat map of zip codes or counties, with each zip code or county being colored by the number of policies located within it. As the policy data contains zip codes this would allow users to visualize that part of the policies better at the cost of less flexibility. This would require changing how the DAG aggregates data as well as how the front end calls data from the back end.

Lessons Learned

We learned new technical skills, including AWS and web development. Using CloudFormation templates to create DynamoDB tables and define AppSync resolvers had a massive learning curve, but proved to be effective in automating the creation of the project structure. On the front end, node.js and React were new, but were easy to pick up and became a strong skill. The iterative process, using Agile in a professional environment, was valuable in helping us realize an effective path to take for larger projects. On the design side, the importance of a working design document was realized as it allowed us to quickly inform other engineers not directly involved on the project what we were doing and how they could help. Finally, github actions and workflows were learned in order to deploy and test the code automatically.

Team Profile

Lucas Bowar - Back end developer on this project; worked on implementing the DynamoDB table, and the DAG.

Sean Keenan - Front end developer on this project; worked on making the map and other UI elements, displaying the H3 tiles in the front end and making the cloudformation template for the front end.

Nicholas Stevenson - Back end developer on this project; worked on the GraphQL schema, AppSync resolvers, DynamoDB table, and integrating front end with back end.

Pouya Zakeri - Front end developer on this project; worked on making the map and other UI elements, displaying the H3 tiles in the front end and making the cloudformation template for the front end.

Appendix

H3 Tiles: Geographical hexagons displayed on a map. Hexagons are different sizes at different resolutions, for example at resolution 1 a hexagon can cover an entire state, and at resolution 8 a hexagon can cover an entire neighborhood. Example H3 tiles shown in Figure 6.

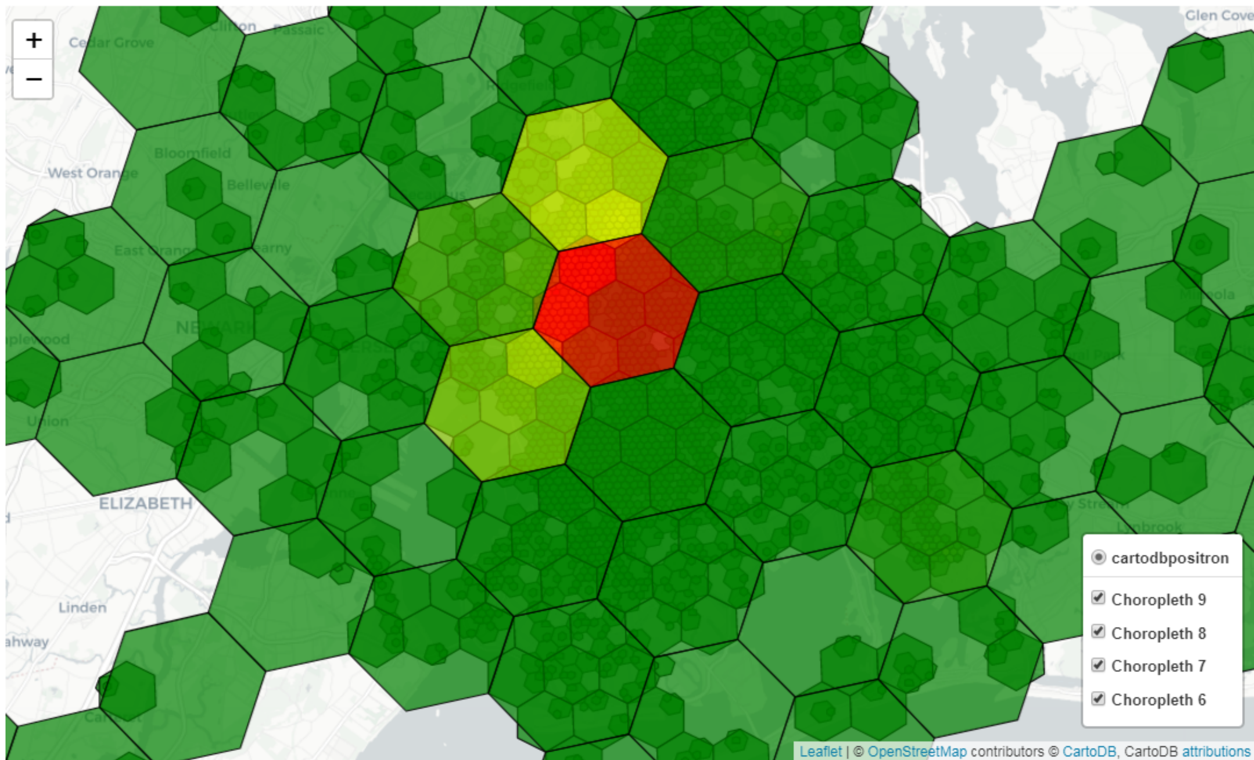

Figure 7: Example H3 Tiles