Neo4j - Uber H3 - Geospatial

We are going to take a slight detour with regards to the healthcare blog series and talk about Uber H3. H3 is a hexagonal hierarchical geospatial indexing system. It comes with an API for indexing coordinates into a global grid. The grid is fully global and you can choose your resolution. The advantages and disadvantages of the hexagonal grid system are discussed here and here . Uber open-sourced the H3 indexing system and it comes with a set of java bindings that we will use with Neo4j.

Since version 3.4, Neo4j has a native geospatial datatype. Neo4j uses the WGS-84 and WGS-84 3D coordinate reference system. Within Neo4j, we can index these point properties and query using our distance function or you query within a bounding box. Max DeMarzi blogged about this as did Neo4j. At GraphConnect 2018, Will Lyon and Craig Taverner had a session on Neo4j and Going Spatial. These are all great resources for Neo4j and geo-spatial search.

Why Uber H3

Uber H3 adds some new features that can be extended through Neo4j procedures. Specifically, we want to be able to query based on a polygon, query based on a polygon with holes, and even along a line. We will use the data from our healthcare demo dataset and show how we can use H3 hexagon addresses to help our queries.

H3 Procedure

For our first procedure, we will pass in the latitude, longitude and resolution and receive a hexAddress back.

The H3 interface also allows us to query via a polygon. With Neo4j, we can query using a bounding box like so:

Neo4j Bounding Box Query

In H3, the polygon search looks like this:

This returns in 17ms against 4.4 million locations.

If we combine this with our healthcare data, we can find providers with a certain taxonomy.

We can make this polygon as complicated as we need. This example is for a county polygon:

```
CALL com.dfauth.h3.polygonSearch([{lon:"-77.302457",lat:"38.504683"},{lon
:"-77.310334",lat:"38.493926"},{lon:"-77.322622",lat:"38.467131"},{lon:"-
77.32544",lat:"38.44885"},{lon:"-77.319036",lat:"38.417803"},{lon:"-77.31
0719",lat:"38.397669"},{lon:"-77.312201",lat:"38.390958"},{lon:"-77.31484
8",lat:"38.389579"},{lon:"-77.317288",lat:"38.383576"},{lon:"-77.296077",
lat:"38.369797"},{lon:"-77.288145",lat:"38.359477"},{lon:"-77.28835",lat:
"38.351286"},{lon:"-77.286202",lat:"38.347025"},{lon:"-77.286202",lat:"38
.347024"},{lon:"-77.321403",lat:"38.345226"},{lon:"-77.339268",lat:"38.30
2723"},{lon:"-77.345728",lat:"38.26139"},{lon:"-77.326692",lat:"38.245136
"},{lon:"-77.370301",lat:"38.246576"},{lon:"-77.39085",lat:"38.245589"},{
```

```
lon:"-77.420148",lat:"38.257986"},{lon:"-77.447126",lat:"38.284614"},{lon
:"-77.455692",lat:"38.301341"},{lon:"-77.467053",lat:"38.31866"},{lon:"-7
7.475137",lat:"38.32096"},{lon:"-77.478996",lat:"38.316693"},{lon:"-77.49
8754",lat:"38.32543"},{lon:"-77.506782",lat:"38.325925"},{lon:"-77.527185
",lat:"38.320655"},{lon:"-77.526243",lat:"38.309531"},{lon:"-77.530289",l
at:"38.309172"},{lon:"-77.54546",lat:"38.325081"},{lon:"-77.584673",lat:"
38.346806"},{lon:"-77.594796",lat:"38.336022"},{lon:"-77.618727",lat:"38.
367835"},{lon:"-77.634835",lat:"38.409713"},{lon:"-77.628433",lat:"38.452
075"},{lon:"-77.634157",lat:"38.464508"},{lon:"-77.568349",lat:"38.520177
"},{lon:"-77.530914",lat:"38.555929"},{lon:"-77.481488",lat:"38.592432"},
{lon:"-77.463949",lat:"38.578686"},{lon:"-77.448683",lat:"38.580792"},{lo
n:"-77.395824",lat:"38.545827"},{lon:"-77.370142",lat:"38.519865"},{lon:"
-77.334902",lat:"38.514569"},{lon:"-77.308138",lat:"38.499699"},{lon:"-77
.310528",lat:"38.505289"},{lon:"-77.302457",lat:"38.504683"}],[{}]) yield
 nodes return nodes
```

In these examples, there are no "donut holes". If I need a polygon donut hole to exclude an area, I can pass it in as the second parameter.

```
CALL com.dfauth.h3.polygonSearch([{lon:"-77.302457",lat:"38.504683"},{lon
:"-77.310334",lat:"38.493926"},{lon:"-77.322622",lat:"38.467131"},{lon:"-
77.32544",lat:"38.44885"},{lon:"-77.319036",lat:"38.417803"},{lon:"-77.31
0719",lat:"38.397669"},{lon:"-77.312201",lat:"38.390958"},{lon:"-77.31484
8",lat:"38.389579"},{lon:"-77.317288",lat:"38.383576"},{lon:"-77.296077",
lat:"38.369797"},{lon:"-77.288145",lat:"38.359477"},{lon:"-77.28835",lat:
"38.351286"},{lon:"-77.286202",lat:"38.347025"},{lon:"-77.286202",lat:"38
.347024"},{lon:"-77.321403",lat:"38.345226"},{lon:"-77.339268",lat:"38.30
2723"},{lon:"-77.345728",lat:"38.26139"},{lon:"-77.326692",lat:"38.245136
"},{lon:"-77.370301",lat:"38.246576"},{lon:"-77.39085",lat:"38.245589"},{
lon:"-77.420148",lat:"38.257986"},{lon:"-77.447126",lat:"38.284614"},{lon
:"-77.455692",lat:"38.301341"},{lon:"-77.467053",lat:"38.31866"},{lon:"-7
7.475137",lat:"38.32096"},{lon:"-77.478996",lat:"38.316693"},{lon:"-77.49
8754",lat:"38.32543"},{lon:"-77.506782",lat:"38.325925"},{lon:"-77.527185
",lat:"38.320655"},{lon:"-77.526243",lat:"38.309531"},{lon:"-77.530289",l
at:"38.309172"},{lon:"-77.54546",lat:"38.325081"},{lon:"-77.584673",lat:"
38.346806"},{lon:"-77.594796",lat:"38.336022"},{lon:"-77.618727",lat:"38.
```

```
367835"},{lon:"-77.634835",lat:"38.409713"},{lon:"-77.628433",lat:"38.452
075"},{lon:"-77.634157",lat:"38.464508"},{lon:"-77.568349",lat:"38.520177
"},{lon:"-77.530914",lat:"38.555929"},{lon:"-77.481488",lat:"38.592432"},
{lon:"-77.463949",lat:"38.578686"},{lon:"-77.448683",lat:"38.580792"},{lo
n:"-77.395824",lat:"38.545827"},{lon:"-77.370142",lat:"38.519865"},{lon:"
-77.334902",lat:"38.514569"},{lon:"-77.308138",lat:"38.499699"},{lon:"-77
.310528",lat:"38.505289"},{lon:"-77.302457",lat:"38.504683"}],[{lon:'-77.
530886',lat:'38.3609911'},{lon:'-77.524492',lat:'38.3411698'},{lon:'-77.5
24492',lat:'38.277433'},{lon:'-77.5731773',lat:'38.2774607'},{lon:'-77.59
4635',lat:'38.2771873'}]) yield nodes return nodes
```

In the 3.3.0 release of the Java bindings, H3 included the ability to return all hex addresses along a line between two points. One could combine this with a service like Google Directions to find all locations along a route. Imagine finding doctors along a route or find all events that occurred along a route.
Here is an example that returns providers who have billing addresses along a line:

```
CALL com.dfauth.h3.lineBetweenLocations(38.418582, -77.385268,38.500603,
-77.444288) yield nodes  unwind nodes as locNode  match (locNode)(t:Taxon
omyCode)  return distinct locNode.Address1 + ' ' + locNode.CityName +', '
 + locNode.StateName as locationAddress, locNode.latitude as latitude, lo
cNode.longitude as longitude, coalesce(p.BusinessName,p.FirstName + ' ' +
 p.LastName) as practiceName, p.NPI as NPI order by locationAddress;
```

Pretty neat, right? As always, the source code as always is available on github.