**Geospatial data handing and Queries in MongoDB**

# Introduction

In this exercise we will learn and explore the Geospatial support in MongoDB document database. MongoDB supports query operations on geospatial data. We will learn how to store geospatial data, create geospatial index, and performing geospatial search using MongoDB geospatial query operators like near, geoWithin, and geoIntersects.

# MongoDB support of Geospatial Data

MongoDB stores data in flexible, JSON-like documents, meaning fields can vary from document to document and data structure can be changed over time. JSON is a flexible format that allows representing data using a semi-structured approach. MongoDB allows storing geospatial data as [GeoJSON](https://docs.mongodb.com/manual/geospatial-queries/index.html#geospatial-geojson) objects or as [legacy coordinate pairs](https://docs.mongodb.com/manual/geospatial-queries/index.html#geospatial-legacy).

**Collections:** MongoDB stores documents in collections. Collections are analogous to tables in relational databases.

* **GeoJSON**

GeoJSON (<https://geojson.org/> ) is a format for encoding a variety of geographic data structures. Itis an open standard format designed for representing simple geographical features, along with their non-spatial attributes. It is based on the JSON format.

*{*

*"type": "Feature",*

*"geometry": {*

*"type": "Point",*

*"coordinates": [78.04373145103455,*

*30.340157821726486]*

*},*

*"properties": {*

*"name": "Geoinformatics Department, IIRS"*

*}*

*}*

GeoJSON supports the following geometry types: Point, LineString, Polygon, MultiPoint, MultiLineString, and MultiPolygon.

# Storing Geospatial Data

MongoDB allows storing geospatial data as GeoJSON objects. Here we will see how to store geospatial data in MongoDB. To specify GeoJSON data, we can use:

* a field named type that specifies the [GeoJSON object type](https://docs.mongodb.com/manual/reference/geojson/) and
* a field named coordinates that specifies the object’s coordinates.

Note: If specifying latitude and longitude coordinates, list the longitude first and then latitude

* + Valid longitude values are between -180 and 180, both inclusive.
  + Valid latitude values are between -90 and 90, both inclusive.

<field>: { type: <GeoJSON type> , coordinates: <coordinates> }

* **Point Type**

Point type represents a single specific point. Following is an example of an object with a location field as a point:

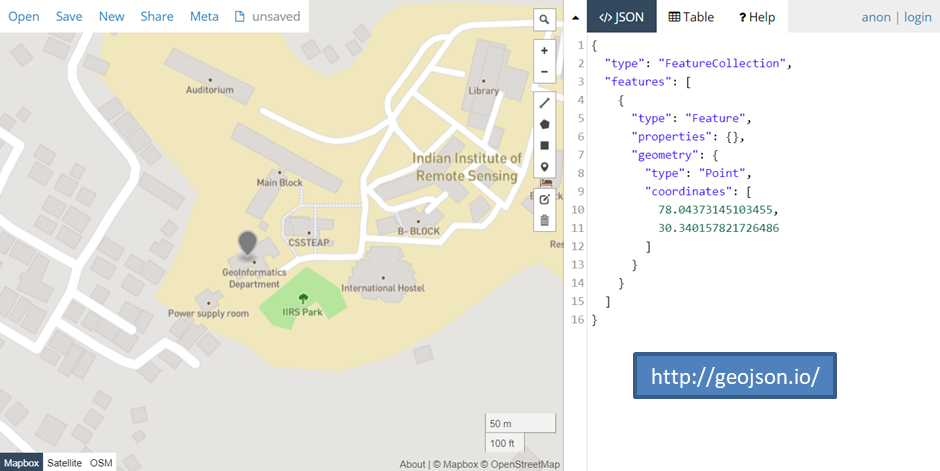
location: {

type: "Point",

coordinates: [78.04373145103455,

30.340157821726486]

}



Note that the longitude’s value comes before the latitude. MongoDB geospatial queries on GeoJSON objects calculate on a sphere and use the WGS84 reference system for geospatial queries on GeoJSON objects.

* **Legacy Coordinate Pairs**

To calculate distances on a Euclidean plane, location data has to be stores as legacy coordinate pairs and use a [2d](https://docs.mongodb.com/manual/geospatial-queries/index.html#geo-2d) index. MongoDB supports spherical surface calculations on legacy coordinate pairs via a [2dsphere](https://docs.mongodb.com/manual/geospatial-queries/index.html#geo-2dsphere) index by converting the data to the GeoJSON Point type. To specify data as legacy coordinate pairs, you can use either an array or an embedded document.

1. **Specify via an array (Preferred):**

**<field>: [ <x>, <y> ]**

Is similar to <field>: [<longitude>, <latitude> ]

1. **Specify via an embedded document:**

**<field>: { <field1>: <x>, <field2>: <y> }**

Note: If specifying latitude and longitude coordinates, the first field, regardless of the field name, must contains the **longitude** value and the second field, the **latitude** value ; i.e. **<field>: { <field1>: <longitude>, <field2>: <latitude> }**

* **Polygon type**

The polygon is the most complex of the GeoJSON types and is used to describe an area. Here is an example of a location defined as a polygon:

"location": {

"type": "Polygon",

"coordinates": [

[

[

78.04366707801817,

30.340038606452104

],

[

78.043722063303,

30.340060597630146

],

[

78.04369926452637,

30.340083746233283

],

[

78.0437596142292,

30.34011615426845

],

[

78.04379716515541,

30.340043236174214

],

[

78.0438856780529,

30.340079116513085

],

[

78.04383605718613,

30.34016592373016

],

[

78.04388031363487,

30.34018560002201

],

[

78.0439031124115,

30.340167081159226

],

[

78.04395005106925,

30.340182127735506

],

[

78.04387092590332,

30.3402747220002

],

[

78.04360136389732,

30.34014624743437

],

[

78.04366707801817,

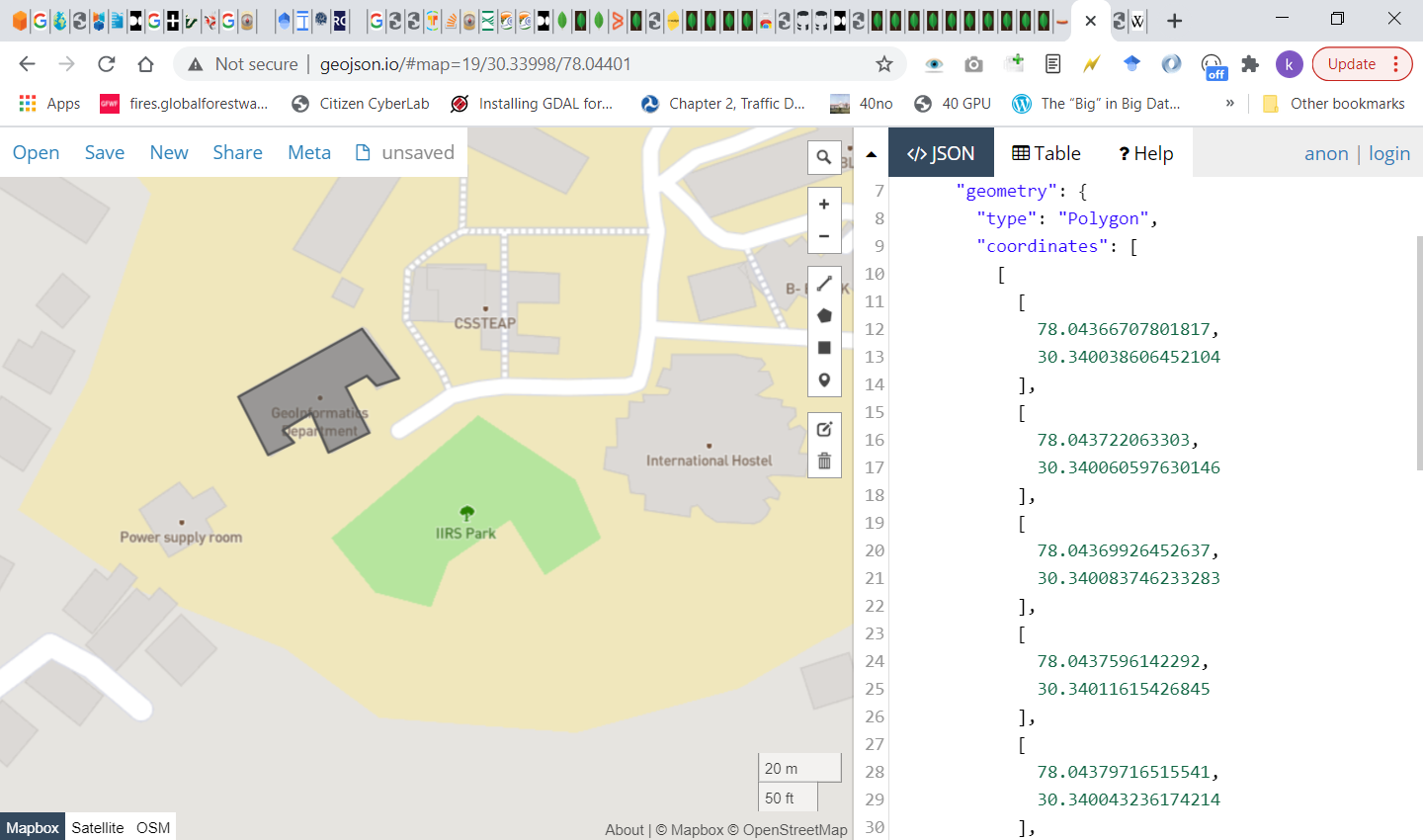
30.340038606452104

]

]

]

}



# Geospatial Indexing

To perform search queries on the geospatial data we stored, we need to create a geospatial index on our location field. MongoDB provides the following geospatial index types to support the geospatial queries.

* **2dsphere**

[2dsphere](https://docs.mongodb.com/manual/core/2dsphere/) indexes support queries that calculate [geometries on an earth-like sphere](https://docs.mongodb.com/manual/geospatial-queries/index.html#geospatial-geometry).

To create a 2dsphere index, the [***db.collection.createIndex()***](https://docs.mongodb.com/manual/reference/method/db.collection.createIndex/#db.collection.createIndex) method has to be used specifying the string literal "2dsphere" as the index type:

**db.collection.createIndex( { <location field> : "2dsphere" } )**

here the <location field> is a field whose value is either a [GeoJSON object](https://docs.mongodb.com/manual/geospatial-queries/index.html#geospatial-geojson) or a [legacy coordinates pair](https://docs.mongodb.com/manual/geospatial-queries/index.html#geospatial-legacy).

* **2d**

[2d](https://docs.mongodb.com/manual/core/2d/) indexes support queries that calculate [geometries on a two-dimensional plane](https://docs.mongodb.com/manual/geospatial-queries/index.html#geospatial-geometry). To create a 2d index, [***db.collection.createIndex()***](https://docs.mongodb.com/manual/reference/method/db.collection.createIndex/#db.collection.createIndex)method is available which takes the location field as the key and the string literal "2d" as the index type:

db.collection.createIndex( { <location field> : "2d" } )

here the <location field> is a field whose value is a [legacy coordinates pair](https://docs.mongodb.com/manual/geospatial-queries/index.html#geospatial-legacy).

# Geospatial Query Operators

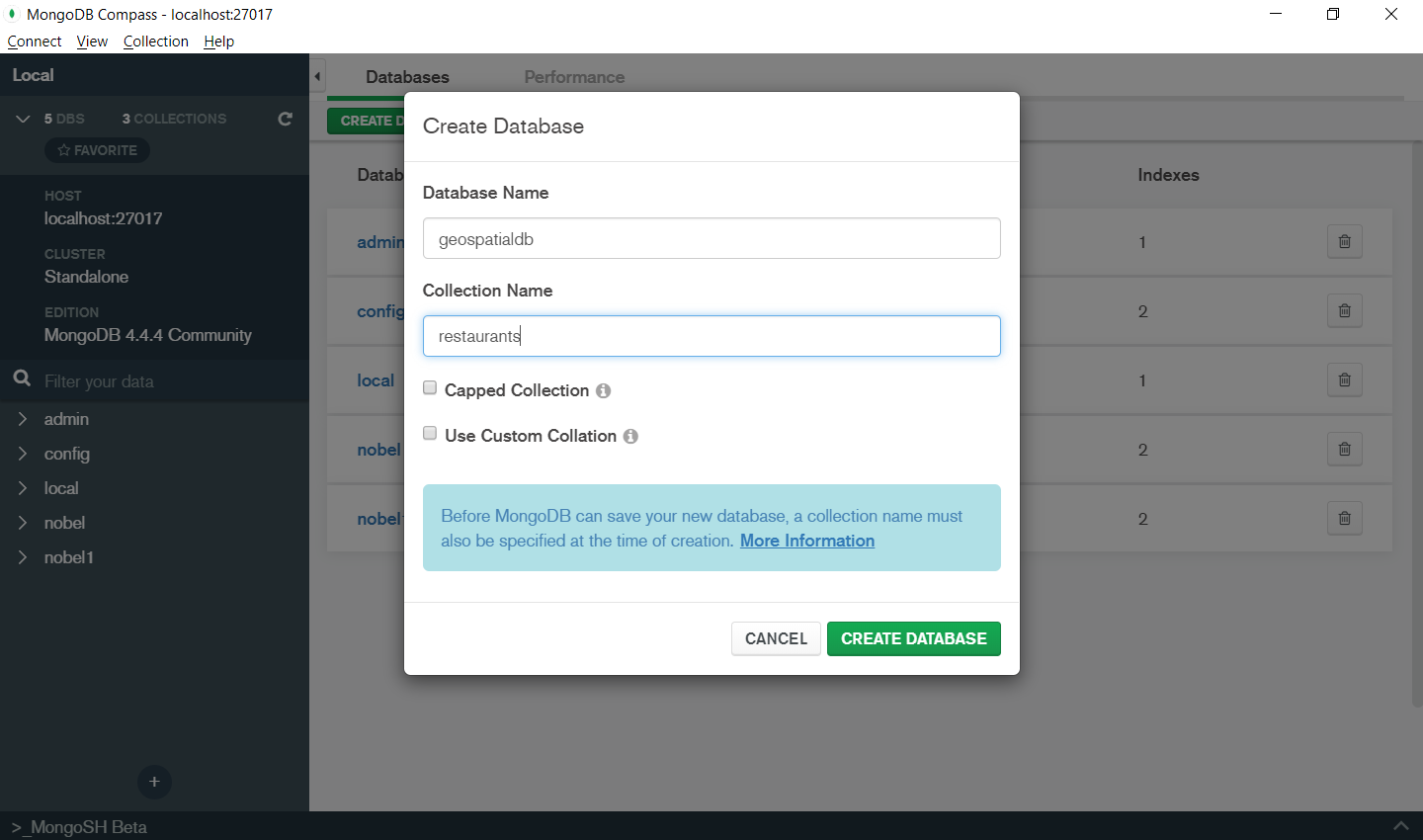
MongoDB provides the following geospatial query operators:

|  |  |
| --- | --- |
| **Name** | **Description** |
| [**$geoIntersects**](https://docs.mongodb.com/manual/reference/operator/query/geoIntersects/#op._S_geoIntersects) | Selects geometries that intersect with a [GeoJSON](https://docs.mongodb.com/manual/reference/glossary/#term-geojson) geometry. The [2dsphere](https://docs.mongodb.com/manual/core/2dsphere/) index supports [$geoIntersects](https://docs.mongodb.com/manual/reference/operator/query/geoIntersects/#op._S_geoIntersects). |
| [**$geoWithin**](https://docs.mongodb.com/manual/reference/operator/query/geoWithin/#op._S_geoWithin) | Selects geometries within a bounding [GeoJSON geometry](https://docs.mongodb.com/manual/reference/geojson/#geospatial-indexes-store-geojson). The [2dsphere](https://docs.mongodb.com/manual/core/2dsphere/) and [2d](https://docs.mongodb.com/manual/core/2d/) indexes support [$geoWithin](https://docs.mongodb.com/manual/reference/operator/query/geoWithin/#op._S_geoWithin). |
| [**$near**](https://docs.mongodb.com/manual/reference/operator/query/near/#op._S_near) | Returns geospatial objects in proximity to a point. Requires a geospatial index. The [2dsphere](https://docs.mongodb.com/manual/core/2dsphere/) and [2d](https://docs.mongodb.com/manual/core/2d/) indexes support [$near](https://docs.mongodb.com/manual/reference/operator/query/near/#op._S_near). |
| [**$nearSphere**](https://docs.mongodb.com/manual/reference/operator/query/nearSphere/#op._S_nearSphere) | Returns geospatial objects in proximity to a point on a sphere. Requires a geospatial index. The [2dsphere](https://docs.mongodb.com/manual/core/2dsphere/) and [2d](https://docs.mongodb.com/manual/core/2d/) indexes support [$nearSphere](https://docs.mongodb.com/manual/reference/operator/query/nearSphere/#op._S_nearSphere). |

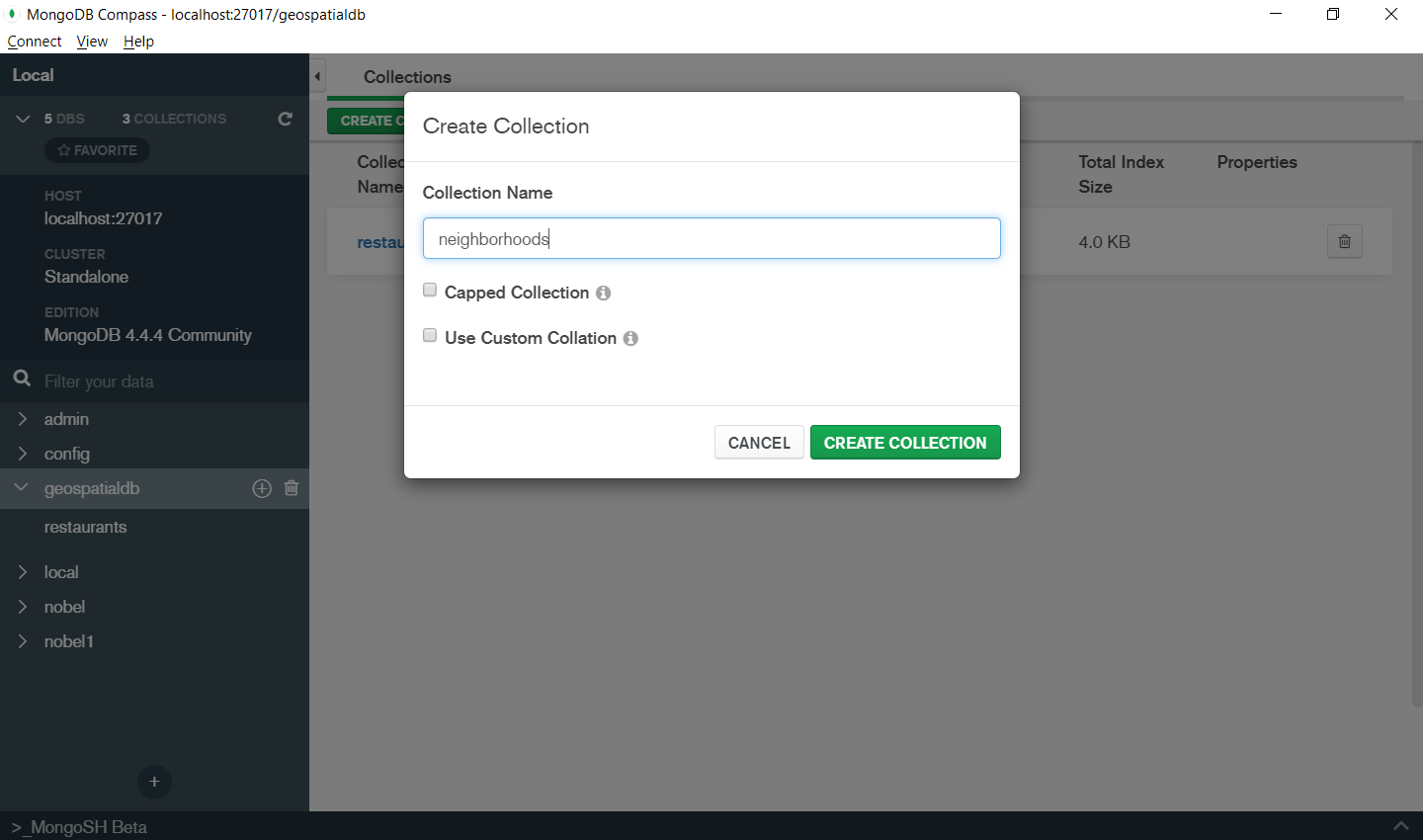
# Geospatial Query using MongoDB

We will use the restaurants and neighborhoods dataset available under the data folder. These contain the data on restaurants and neighborhoods respectively.

Next we will import them into the database (geospatialdb) and storing them in their respective collection. We will use MongoDB compass for importing and creating the database. In compass, click on the “+ Create Database” button available at the bottom of the left panel of the window.

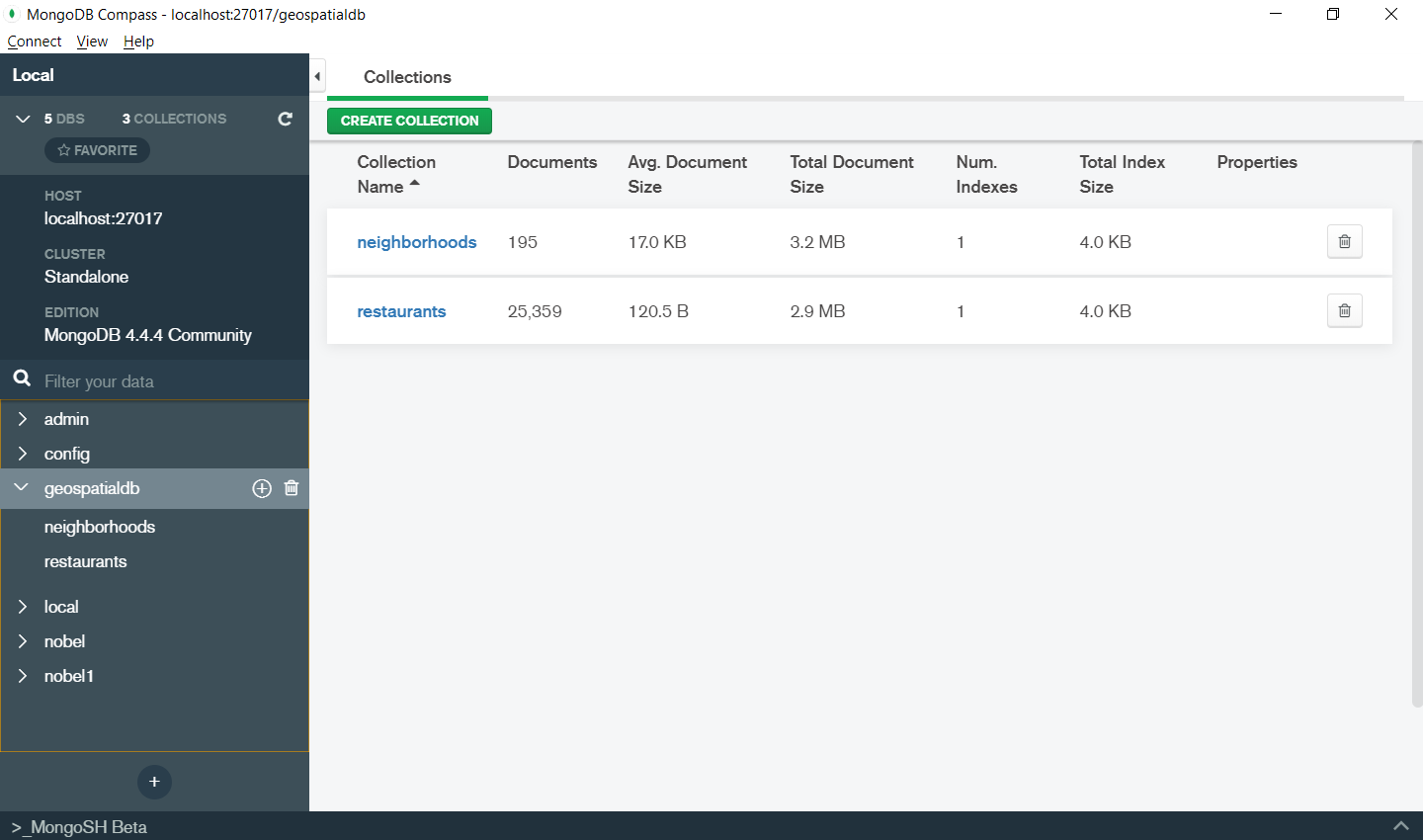


Next add the neighbourhood collection by clicking on the “Create Collection” button



Now we need to import the two json files in their respective collections. We can use the MongoDB compass “Add Data”->”Import File” Option.

Once the data are uploaded we can see their basic description.



* Note the mongoimport utility could also be used using the following command

C:\Program Files\MongoDB\Tools\100\bin

**>mongoimport --db geospatialdb1 --collection restaurants --file d:\data\restaurants.json**

**>mongoimport --db geospatialdb1 --collection neighborhoods --file d:\data\ neighborhoods.json**

A geospatial index improves performance of MongoDB geospatial query operators like [$geoWithin](https://docs.mongodb.com/manual/reference/operator/query/geoWithin/#op._S_geoWithin) and [$geoIntersects](https://docs.mongodb.com/manual/reference/operator/query/geoIntersects/#op._S_geoIntersects). Since our data is geographical, we will create a 2dsphere index on each collection using the [mongo](https://docs.mongodb.com/manual/reference/program/mongo/#bin.mongo) shell:

**db.restaurants.createIndex({ location: "2dsphere" })**

**db.neighborhoods.createIndex({ geometry: "2dsphere" })**

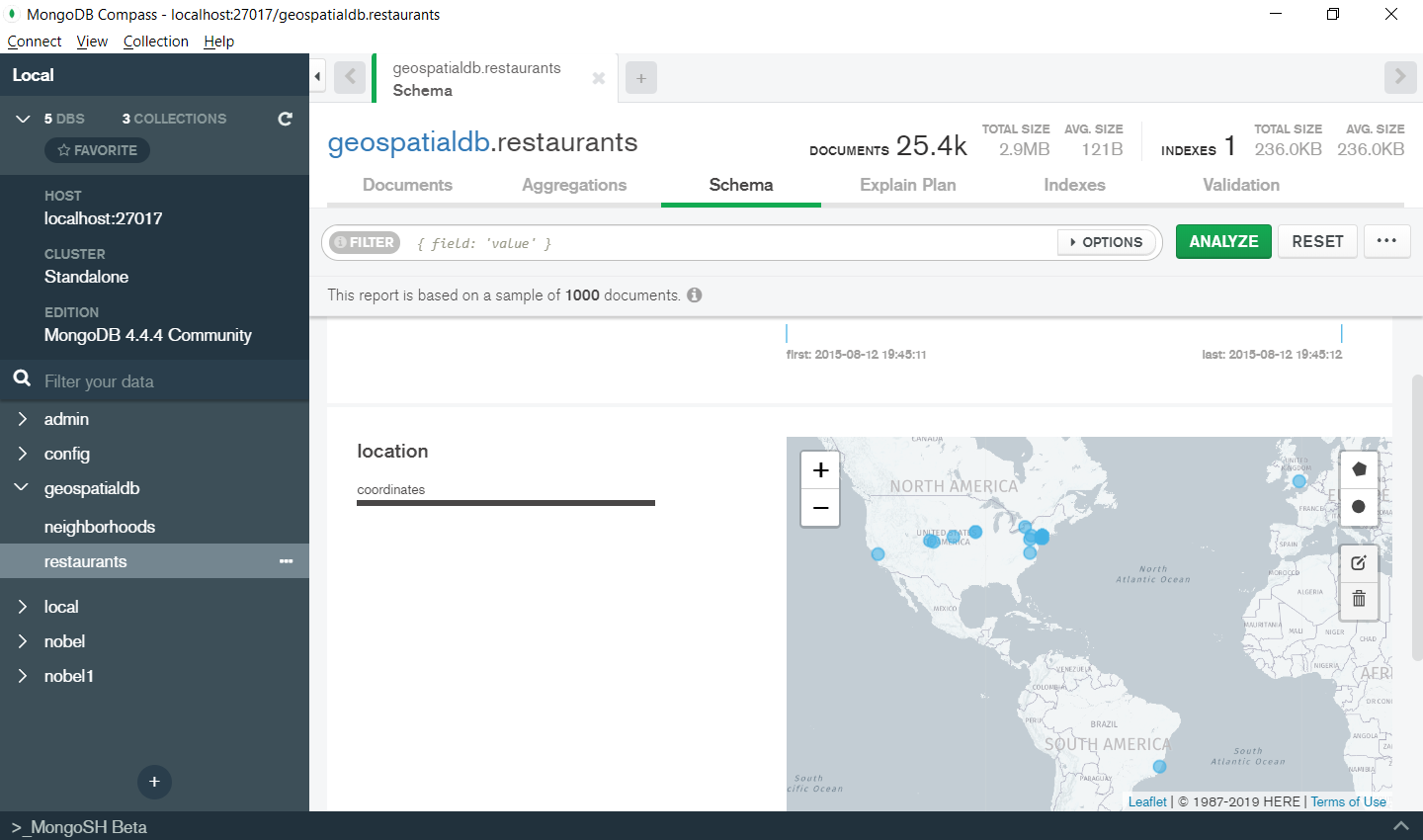
# MongoDB- Exploring the Data

We can inspect an entry in the newly-created restaurants collection from within the [mongo](https://docs.mongodb.com/manual/reference/program/mongo/#bin.mongo) shell:

***use geospatialdb***

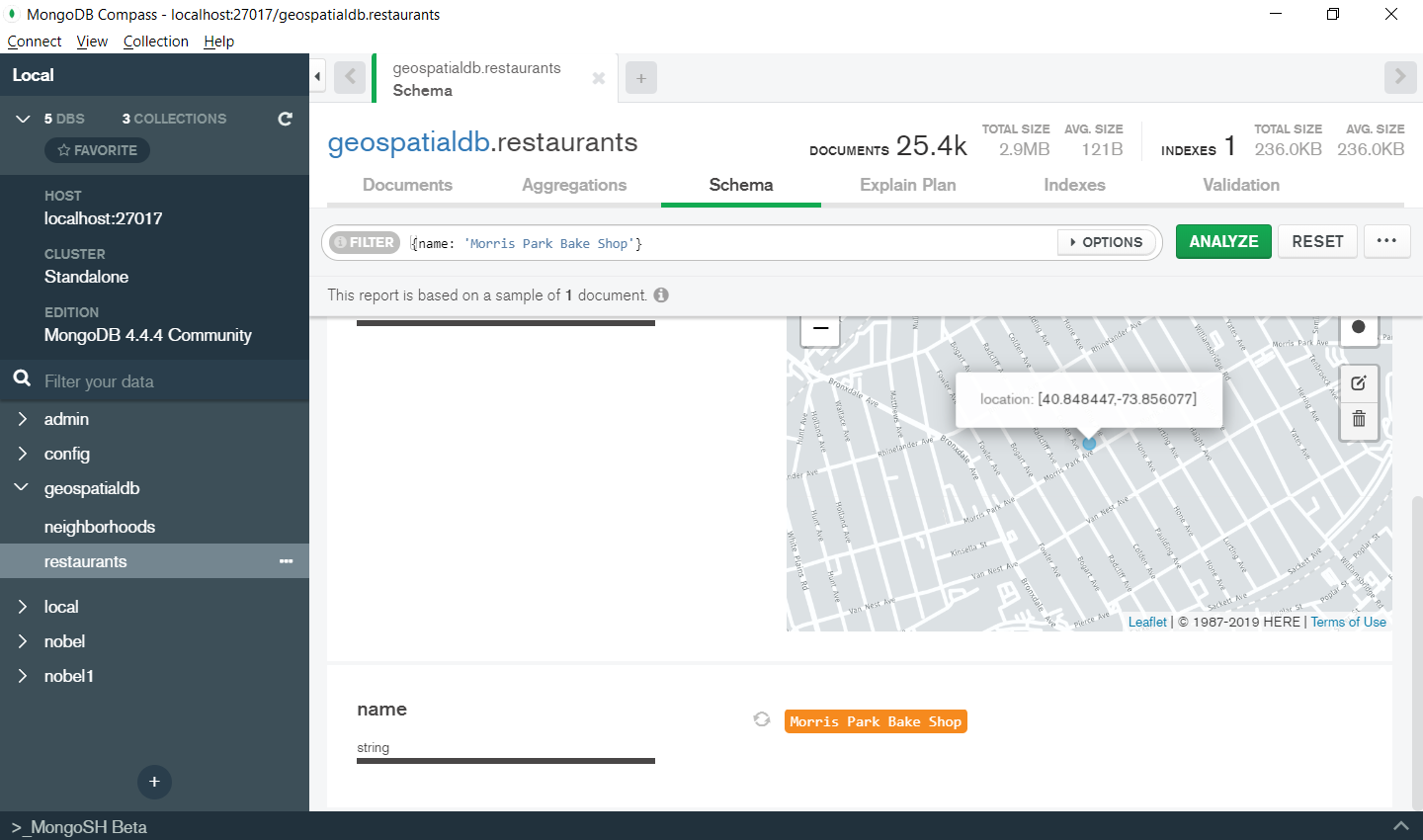
***db.restaurants.findOne()***

MongoDB compass can also be used for exploring the data using its ‘Schema’ tab. In case of geospatial data it automatically shows the map interface with feature to draw and query. Open the schema tab. And click on “Analyse Schema” button.



Next enter the filter as shown in figure

{"name":"Morris Park Bake Shop"}



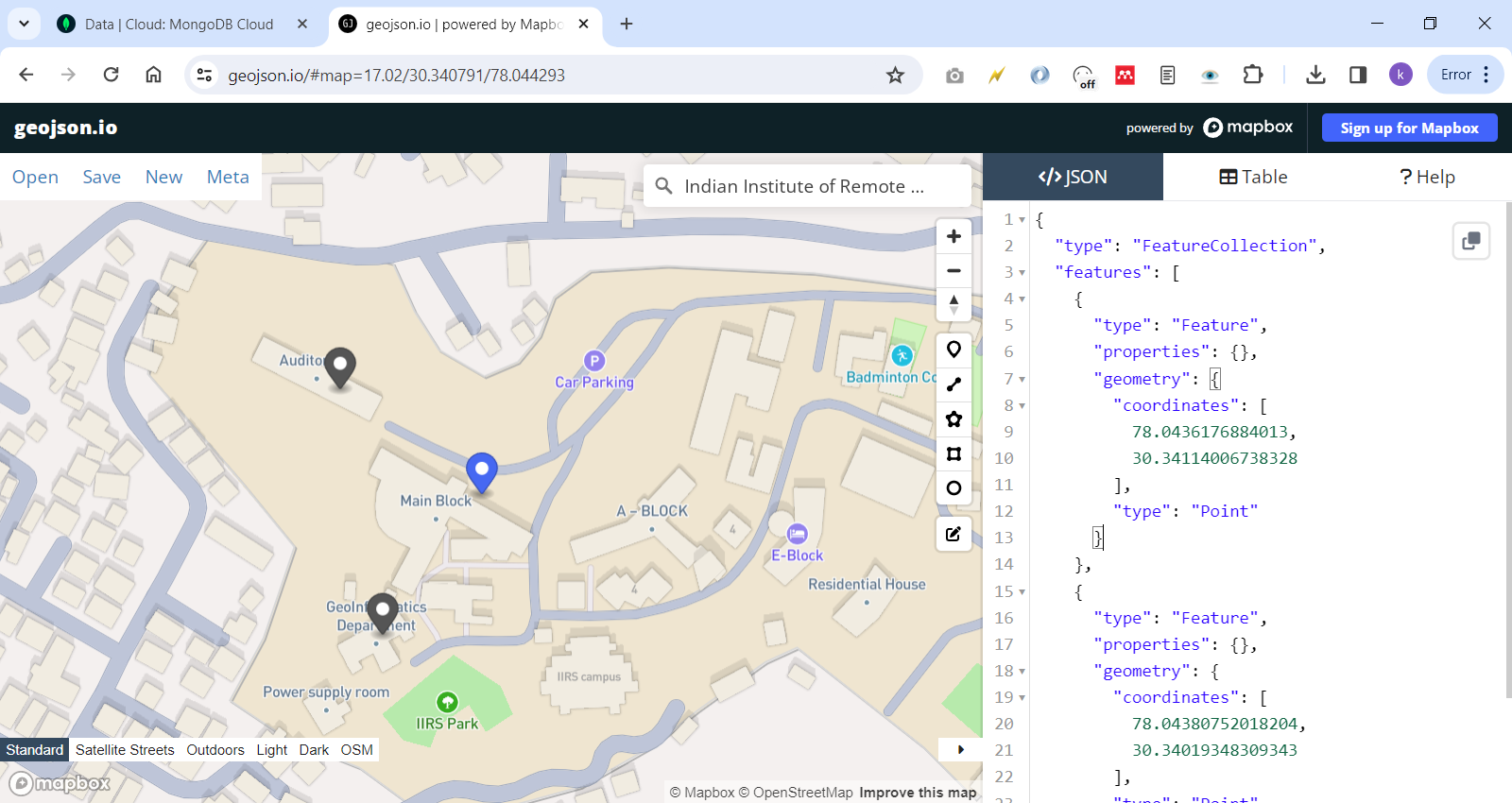
Later we will see how we can also run geospatial query using the schema query builder tool.

Now let’s inspect an entry in the neighborhoods collection we can use the findOne() function:

db.neighborhoods.findOne()

# Create own GeoJSON data and import to MongoDB

Use the website <https://geojson.io/> to create online the GeoJSON data on the IIRS campus building



And insert it as documents in mongodb using compass.

Note: you need to store the individual features not entire feature collection

# Geospatial Query using MongoDB

**Find the Current Neighborhood**

Suppose we know the location of the user (-73.93414657 (longitude) and 40.82302903 (latitude)) and we need to find the current neighbourhood in which the user is located.

To answer this type of queries we can use [**$geoIntersects**](https://docs.mongodb.com/manual/reference/operator/query/geoIntersects/#op._S_geoIntersects)MongoDB operator.

The $geoIntersects operator uses the $geometry operator to specify the GeoJSON object. The syntax is:

{

<location field>: {

$geoIntersects: {

$geometry: {

type: "<GeoJSON object type>" ,

coordinates: [ <coordinates> ]

}

}

}

}

Now to find the neighbourbood in which the user is located i.e. (-73.93414657 longitude and 40.82302903 latitude) we can write the following query

**db.neighborhoods.findOne(**

**{ geometry: { $geoIntersects: { $geometry: { type: "Point",**

**coordinates: [ -73.93414657, 40.82302903 ] } } } }**

**)**

This query will return the following result:

{

"\_id" : ObjectId("55cb9c666c522cafdb053a68"),

"geometry" : {

"type" : "Polygon",

"coordinates" : [

[

[

-73.93383000695911,

40.81949109558767

],

...

]

]

},

"name" : "Central Harlem North-Polo Grounds"

}

**Find all POIs (Restaurants) in the Neighborhood**

Using MongoDb you can also query to find the number of restaurants contained in a given neighborhood. Here we will use the **$geoWithin** geospatial query operator.

**$geoWithin:** This operator selects documents with geospatial data that exists entirely within a specified shape.

The specified shape can be a GeoJSON Polygon, a GeoJSON MultiPolygon, or a shape defined by legacy coordinate pairs. The $geoWithin operator uses the $geometry operator to specify the GeoJSON object.

{

<location field>: {

$geoWithin: {

$geometry: {

type: <"Polygon" or "MultiPolygon"> ,

coordinates: [ <coordinates> ]

}

}

}

}

**Within the neighbourhood query**

Run the following query in the [mongo](https://docs.mongodb.com/manual/reference/program/mongo/#bin.mongo) shell to find the neighborhood containing the user location, and to find the total number of the restaurants within that neighbourhood using count function:

***var neighborhood = db.neighborhoods.findOne( { geometry: { $geoIntersects: { $geometry: { type: "Point", coordinates: [ -73.93414657, 40.82302903 ] } } } } )***

***db.restaurants.find( { location: { $geoWithin: { $geometry: neighborhood.geometry } } } ).count()***

This query will return the result as 127 i.e. there are 127 restaurants in the requested neighbourhood.

**Difference between $geoWithin and $geoIntersects**

The $geoWithin operator is similar to $geoIntersects. For the query to find "documents in region (bounding box)" we could use either $geoIntersects or $geoWithin and get the similar result. However the $geoWithin operator and the $geoIntersects operators have 2 key differences:

* $geoWithin searches for geometries that are contained entirely within a given geometry, whereas $geoIntersects looks for geometries that intersect. In case of geometry contain the points, these are the same, but the results may differ if documents contain polygons type not point type.
* $geoWithin supports several shape operators in addition to $geometry, including $centerSphere, which is the way for performing proximity query like "find all restaurants within 2KM of user location". This we will see now.

**Find POIs (restaurants) within a Distance**

To find restaurants within a specified distance of a point, MongoDB provide two geospatial query operators:

* [**$geoWithin**](https://docs.mongodb.com/manual/reference/operator/query/geoWithin/#op._S_geoWithin) with [**$centerSphere**](https://docs.mongodb.com/manual/reference/operator/query/centerSphere/#op._S_centerSphere) to return results in unsorted order, or
* **$nearSphere** with [**$maxDistance**](https://docs.mongodb.com/manual/reference/operator/query/maxDistance/#op._S_maxDistance) if results need to be returned in sorted order based on the distance (like nearest to farthest).

**Unsorted with $geoWithin**

To find restaurants within a circular region (proximity query), $geoWithin with $centerSphere can be used. $centerSphere is a MongoDB-specific syntax to denote a circular region by specifying the center and the radius in radians.

Note: [$geoWithin](https://docs.mongodb.com/manual/reference/operator/query/geoWithin/#op._S_geoWithin) does not return the documents in any specific order, so it may show the user the furthest documents first.

The following will find all restaurants within five kilometre of the user location:

***db.restaurants.find({ location:***

***{ $geoWithin:***

***{ $centerSphere: [ [ -73.93414657, 40.82302903 ], 5 / 6378.1] } } })***

centrosphere’s second argument accepts the radius in radians, so you must divide it by the radius of the earth in miles.

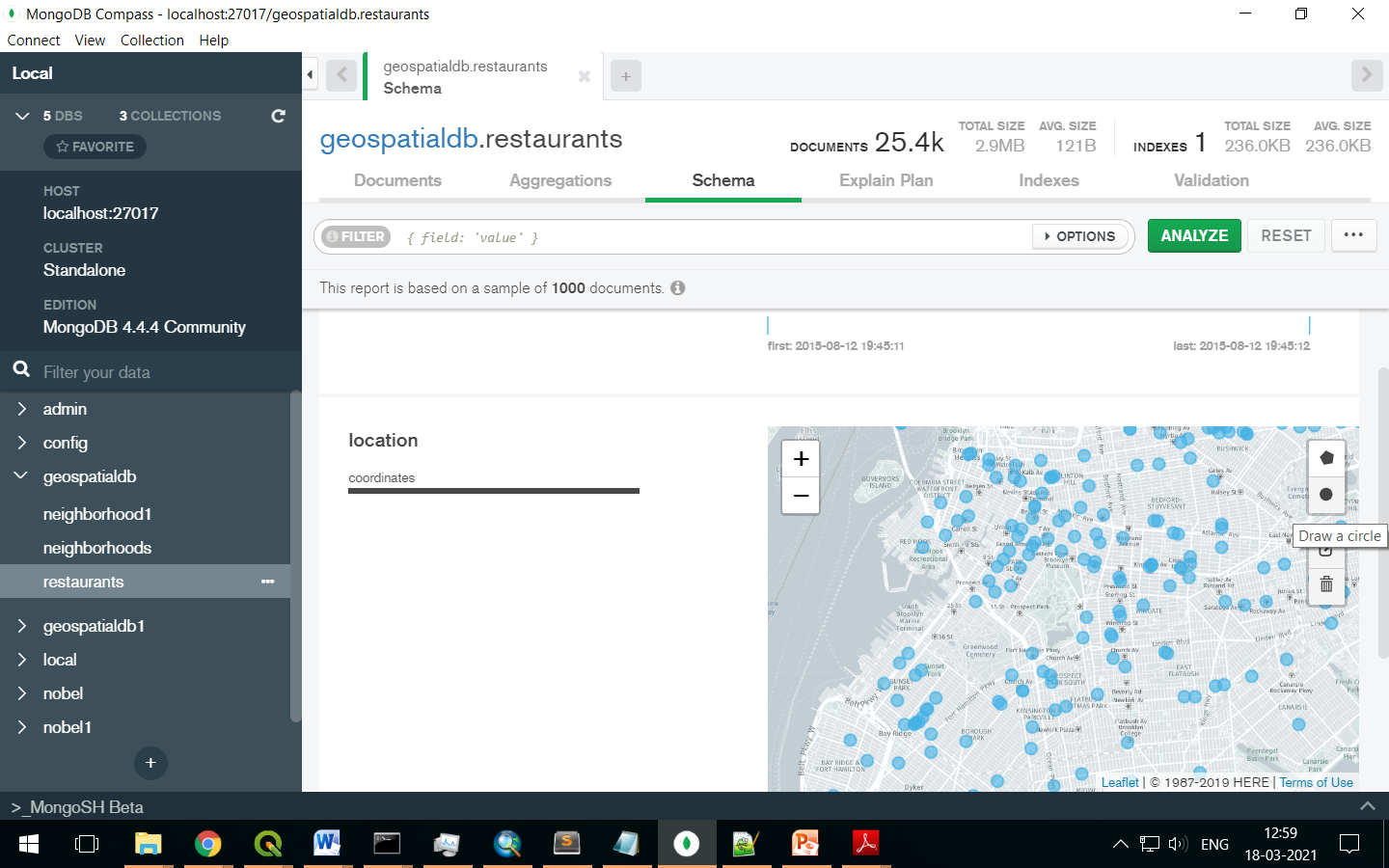
Why “6378.1” is used ? $centerSphere takes the distance in radians, so you need to divide by the radius of the Earth, which is about 6378.1 kilometre at the equator.

**Note:**

* For spherical query operators to function properly, convert distances to radians, and convert from radians to the distances units used by your application.
* The equatorial radius of the Earth is approximately 3,963.2 miles or 6,378.1 kilometres.
* To convert:
* **distance to radians**: divide the distance by the radius of the sphere (e.g. the Earth) in the same units as the distance measurement.
* **radians to distance**: multiply the radian measure by the radius of the sphere (e.g. the Earth) in the units system that you want to convert the distance to.
* 1 miles (mi) = 1.609344 km.

#### MongoDB Compass- Map and Geo Query Features

Compass also provides the ability to see and understand geographical data. Geo data can be displayed on an interactive map, and drawing on the map creates a geo-query with resulting coordinate pairs displayed on the map. The map can be zoomed and panned with a click and hold of the mouse. The schema tab provides this functionality.

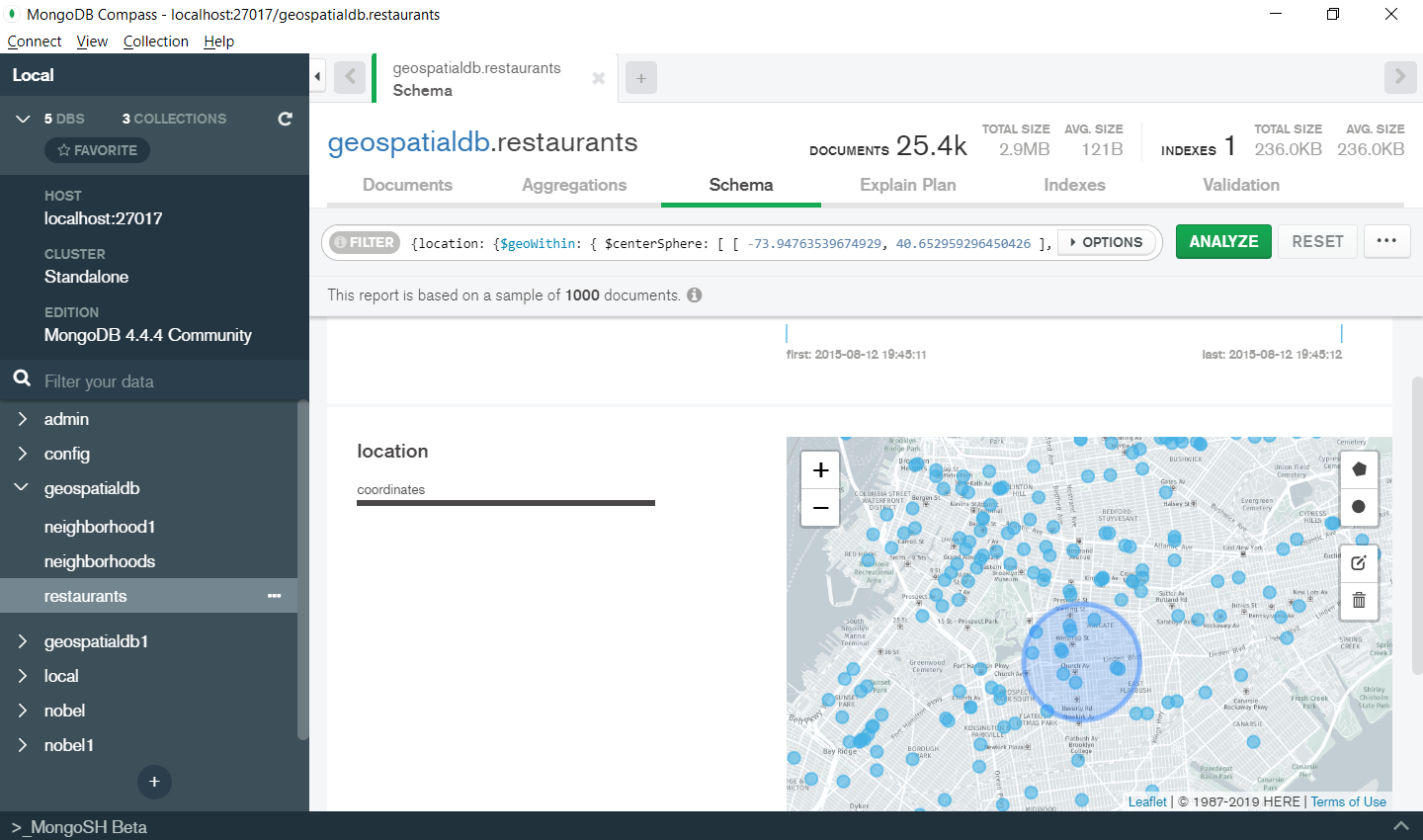


### Apply a Location Filter

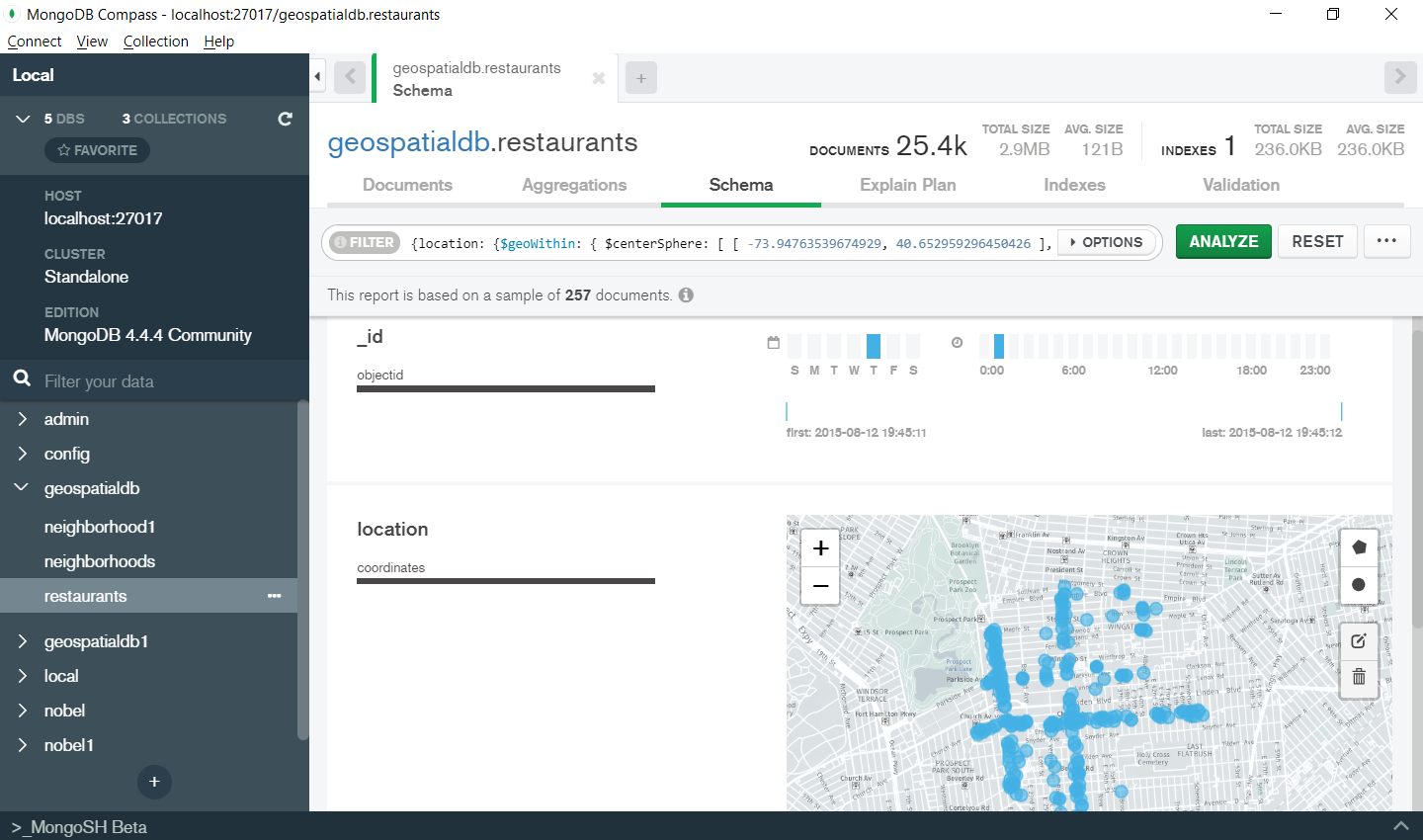
You can apply a filter to the map to only analyse a specific range of points. To define a location filter:

1. Click the Circle or polygon button at the top-right of the map.
2. Click and drag on the map to draw a circle or draw the polygon
3. This specifies the area of the map you want to analyse.
4. Repeat this process as desired to include additional areas of the map in the schema analysis.

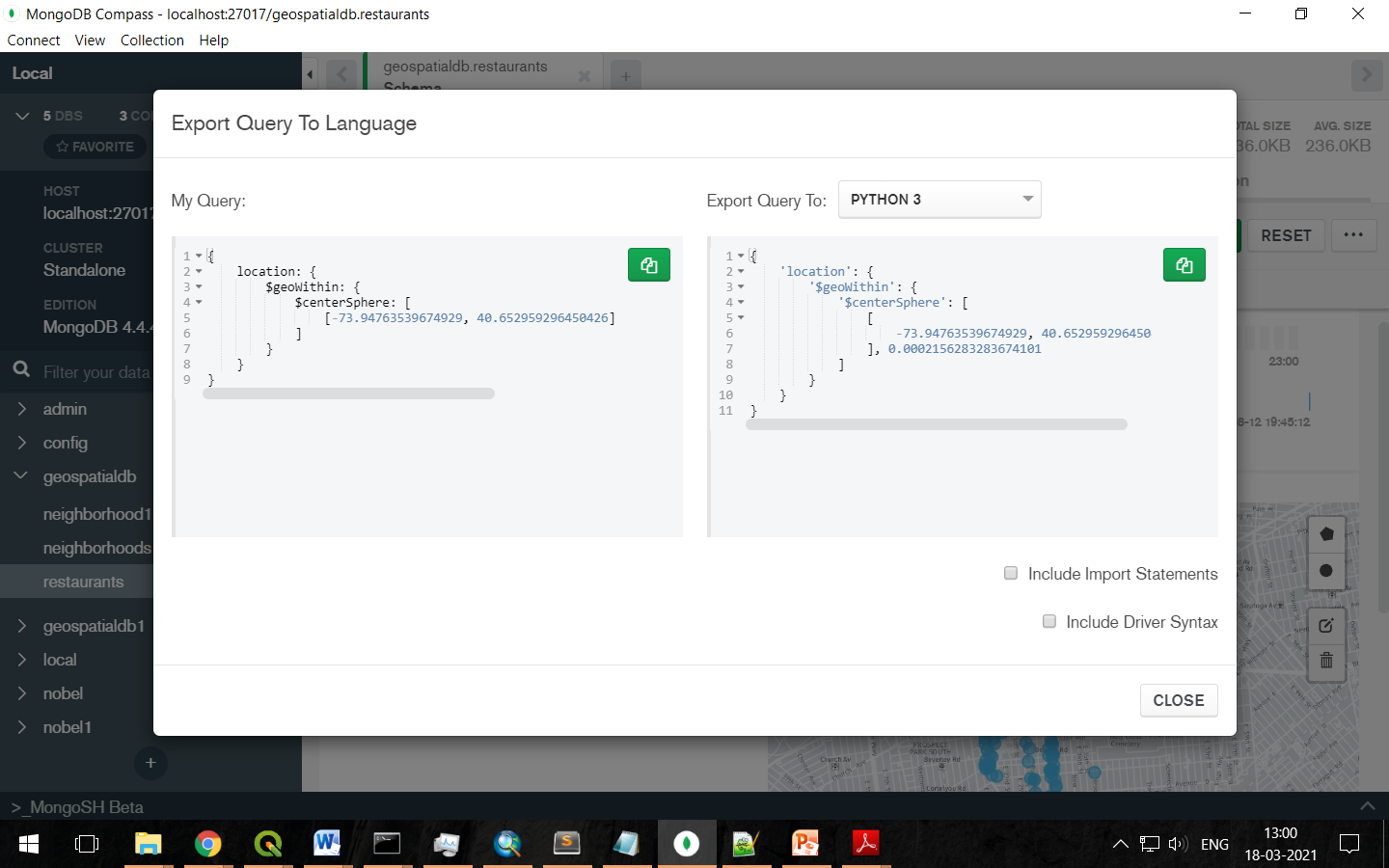
The query bar updates as one draw location filters to show the exact coordinates used in the $geoWithin query applied to the schema analysis. If you specify multiple location filters, the query becomes an $or query with multiple $geoWithin operators.



Clicking on analyse button will show the result of the query



The interactively generated query could also be exported using the “Export Query to Language” option Query



***db.restaurants.find({location: {$geoWithin: { $centerSphere: [ [ -73.94763539674929, 40.652959296450426 ], 0.0002156283283674101 ]}}})***

**Getting Sorted result with $nearSphere**

For getting the sorted result we can also use [$nearSphere](https://docs.mongodb.com/manual/reference/operator/query/nearSphere/#op._S_nearSphere) geospatial query operator and specify a [$maxDistance](https://docs.mongodb.com/manual/reference/operator/query/maxDistance/#op._S_maxDistance) properties in meters. The $nearSphere operator takes a $geometry property which is GeoJSON point, and $minDistance and $maxDistance properties that are in meters.

**$nearSphere Operator:** Specifies a point for which a [geospatial](https://docs.mongodb.com/manual/reference/glossary/#term-geospatial) query returns the documents from nearest to farthest. MongoDB calculates distances for [$nearSphere](https://docs.mongodb.com/manual/reference/operator/query/nearSphere/#op._S_nearSphere) using spherical geometry. [$nearSphere](https://docs.mongodb.com/manual/reference/operator/query/nearSphere/#op._S_nearSphere) *requires* a geospatial index:

* [2dsphere](https://docs.mongodb.com/manual/core/2dsphere/) index for location data defined as GeoJSON points
* [2d](https://docs.mongodb.com/manual/core/2d/) index for location data defined as legacy coordinate pairs.

The [$nearSphere](https://docs.mongodb.com/manual/reference/operator/query/nearSphere/#op._S_nearSphere) operator can specify either a [GeoJSON](https://docs.mongodb.com/manual/reference/glossary/#term-geojson) point or legacy coordinate point. To specify a [GeoJSON Point](https://docs.mongodb.com/manual/reference/geojson/#geojson-point), use the following syntax:

{

$nearSphere: {

$geometry: {

type : "Point",

coordinates : [ <longitude>, <latitude> ]

},

$minDistance: <distance in meters>,

$maxDistance: <distance in meters>

}

}

* The *optional* [$minDistance](https://docs.mongodb.com/manual/reference/operator/query/minDistance/#op._S_minDistance) limits the results to those documents that are *at least* the specified distance from the center point.
* The *optional* [$maxDistance](https://docs.mongodb.com/manual/reference/operator/query/maxDistance/#op._S_maxDistance) is available for specifying the farthest distance. It limits the results to those documents that are not greater than the specified distance from the center point.

To specify a point using legacy coordinates, use the following syntax:

{

$nearSphere: [ <x>, <y> ],

$minDistance: <distance in radians>,

$maxDistance: <distance in radians>

}

* The optional $minDistance is available only if the query uses the 2dsphere index. $minDistance limits the results to those documents that are at least the specified distance from the center point.
* The optional $maxDistance is available for either index.
* If you use longitude and latitude for legacy coordinates, specify the longitude first, then latitude.

Now suppose we want to find all restaurants within 5 Km of the user location in sorted order from nearest to farthest we can write the query as:

***db.restaurants.find({ location: { $nearSphere: { $geometry: { type: "Point", coordinates: [ -73.93414657, 40.82302903 ] }, $maxDistance: 5000 } } })***

**Geospatial Aggregation Stage**

MongoDB provides the **$geoNear** geospatial aggregation pipeline stage. It returns an ordered stream of documents based on the proximity to a geospatial point i.e. it outputs documents in order of nearest to farthest from a specified point. It also incorporates the functionality of $sort, $limit etc. for geospatial data.

The output documents include an additional distance field and can include a location identifier field. $geoNear requires a geospatial index.

Syntax:

{ $geoNear: { <geoNear options> } }

$geoNear is an aggregation framework stage that takes in several properties:

* **near**, which is the GeoJSON point (The point for which to find the closest documents.)
* **spherical**, which must be true if you're using a 2dsphere index (and you should use a 2dsphere index rather than a 2d index unless you have a good reason)
* **maxDistance** and **minDistance**, in meters
* **distanceField** and **distanceMultiplier**, which are the field to put the computed distance in and the factor to multiply it by.

Kindly refer the link [https://docs.mongodb.com/manual/reference/operator/ aggregation/geoNear/#pipe.\_S\_geoNear](https://docs.mongodb.com/manual/reference/operator/%20aggregation/geoNear/#pipe._S_geoNear) for knowing more details

Now if we want to find all restaurants within 1 Km of the user location in sorted order from nearest to farthest along with the distance we can write the query as:

***db.restaurants.aggregate([{***

***$geoNear: {***

***near: {***

***type: 'Point',***

***coordinates: [-73.93414657, 40.82302903]***

***},***

***spherical: true,***

***maxDistance: 1000,***

***distanceField: 'distanceFromuser'***

***}}***

***])***

We can further sort it in descending order using the sort() operator and limiting result to top 5

***db.restaurants.aggregate([{***

***$geoNear: {***

***near: {***

***type: 'Point',***

***coordinates: [-73.93414657, 40.82302903]***

***},***

***spherical: true,***

***maxDistance: 1000,***

***distanceField: 'distanceFromuser'***

***}}, { $sort: { distanceFromuser: -1} }, { $limit: 5 }***

***])***

**Assignment:**

MongoDb Atlas- Create an account for the cloud based Mongodb Product named ‘MongoDb Atlas’. State the important features and advantages