# Lecture Notes for Lecture 13 of CS 5200 (Database Management System) for the Summer 1, 2019 session at the Northeastern University Silicon Valley Campus.

## *MongoDB Geospatial Indexing and Queries*

Philip Gust,
Clinical Instructor
Department of Computer Science

Material contained in this presentation is based in part on and uses content from recommended readings for the course.

# Lecture 11 Review

- In this lecture we looked at techniques for data modeling in MongoDB, including document structure, atomicity of write operations, document growth, and data use and performance.
- Next, we looked at how to perform queries on documents in both the Mongo shell and the Java APIs, and how to control the order and which fields are returned.
- Finally, we discuss several options for graphical development tools to develop MongoDB databases and applications.

# MongoDB Geospatial Indexing

- Representing and querying geospatial information has become an important requirement for a variety of applications. MongoDB is a popular choice for them because of its geospatial capabilities.
- This lecture will briefly introduce the concepts of geospatial indexes, and then look at the two types of geospatial indexing provided by MongoDB: flat(2d) and spherical (2dspherical) and the uses for each of them.
- Next It will cover how geospatial data is represented by the widely used GeoJSON extension to JSON, and the the specific types of geometrical constructs for single and composite shapes
- This lecture will also present the MongoDB operators that allow querying geospatial and look at how to combine the operators.
- Finally, we will look at a simple geospatial application that locates restaurants to see how the pieces fit together.

# MongoDB Geospatial Indexing

- MongoDB's geospatial indexing allows you to efficiently execute spatial queries on a collection that contains geospatial shapes and points.
- Before storing your location data and writing queries, you must decide the type of surface to use to perform calculations.
- The type you choose affects how you store data, what type of index to build, and the syntax of your queries.

  **Flat (2d)**

  - To calculate distances on a Euclidean plane, store your location data as legacy coordinate pairs and use a 2d index.

  **Spherical (2dsphere)**

  - To calculate geometry over an Earth-like sphere, store your location data on a spherical surface and use 2dsphere index.
  - Store your location data as GeoJSON objects with the coordinate-axis order: longitude, latitude. GeoJSON uses the WGS84 datum coordinate system.

# MongoDB Geospatial Indexing

**2d**

- 2d indexes support:
  - Calculations using flat geometry
  - Legacy coordinate pairs (i.e., geospatial points on a flat coordinate system)
  - Compound indexes with only one additional field, as a suffix of the 2d index field

# MongoDB Geospatial Indexing

**2dshere**

- 2dsphere indexes support:
  - Calculations on a sphere
  - GeoJSON objects and include backwards compatibility for legacy coordinate pairs
  - Compound indexes with scalar index fields (i.e. ascending or descending) as a prefix or suffix of the 2dsphere index field

# MongoDB Geospatial Indexing

**Flat vs. Spherical Geometry**

- Geospatial queries can use either flat or spherical geometries, depending on both the query and the type of index in use.
- *2dsphere* indexes support only spherical geometries, while *2d* indexes support both flat and spherical geometries.
- However, queries using spherical geometries will be more performant and accurate with a *2dsphere* index, so you should always use *2dsphere* indexes on geographical geospatial fields.

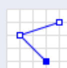
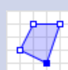
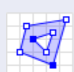
# MongoDB Geospatial Indexing

**GeoJSON Geospatial Format**

- GeoJSON is an open standard format designed for representing simple geographical features, along with their non-spatial attributes. It is based on JSON, the JavaScript Object Notation.
- Features include
  - points (therefore addresses and locations)
  - line strings (therefore streets, highways, boundaries)
  - polygons (countries, provinces, tracts of land)
  - multi-part collections of these types

# MongoDB Geospatial Indexing

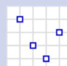
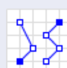
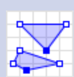
## GeoJSON: Geometry Primitives

| Type | | Examples |
|---|---|---|
| Point |  | { "type": "Point",<br>"coordinates": [30, 10]<br>} |
| LineString |  | { "type": "LineString",<br>"coordinates": [<br>[30, 10], [10, 30], [40, 40] ]<br>} |
| Polygon |  | { "type": "Polygon",<br>"coordinates": [<br>[ [30, 10], [40, 40], [20, 40], [10, 20], [30, 10] ]<br>]<br>} |
| |  | { "type": "Polygon",<br>"coordinates": [<br>[ [35, 10], [45, 45], [15, 40], [10, 20], [35, 10] ],<br>[ [20, 30], [35, 35], [30, 20], [20, 30] ]<br>]<br>} |

Source: Wikipedia

# MongoDB Geospatial Indexing

## GeoJSON: Multi-part Geometries

| Type | | Examples |
|---|---|---|
| MultiPoint |  | { "type": ”MultiPoint",<br>"coordinates": [ [10, 40], [40, 30], [20, 20], [30, 10] ]<br>} |
| MultiLineString |  | { "type": ”MultiLineString",<br>"coordinates": [ [ [10, 10], [20, 20], [10, 40] ],<br>[ [40, 40], [30, 30], [40, 20], [30, 10] ] ]<br>} |
| MultiPolygon |  | { "type": ”MultiPolygon",<br>"coordinates": [ [ [ [30, 20], [45, 40], [10, 40], [30, 20] ] ],<br>[ [ [15, 5], [40, 10], [10, 20], [5, 10], [15, 5] ] ]<br>]<br>} |
| |  | { "type": "Polygon",<br>"coordinates": [ [ [ [40, 40], [20, 45], [45, 30], [40, 40] ] ],<br>[ [ [20, 35], [10, 30], [10, 10], [30, 5], [45, 20], [20, 35] ],<br>[ [30, 20], [20, 15], [20, 25], [30, 20] ] ]<br>]<br>} |

Source: Wikipedia

# MongoDB Geospatial Indexing

**MongoDB Geospatial Query Operators**

- MongoDB's geospatial query operators let you query for:

  **Inclusion**

  - MongoDB can query for locations contained entirely within a specified polygon. Inclusion queries use the $*geoWithin* operator.
  - Both 2d and 2dsphere indexes can support inclusion queries. MongoDB does not require an index for inclusion queries; however, such indexes will improve query performance.

  **Intersection**

  - MongoDB can query for locations that intersect with a specified geometry. These queries apply only to data on a spherical surface. These queries use the $*geoIntersects* operator. Only 2dsphere indexes support intersection.

  **Proximity**

  - MongoDB can query for the points nearest to another point. Proximity queries use the $near operator. The $*near* operator requires a 2d or 2dsphere index.

# MongoDB Geospatial Indexing

## MongoDB Geospatial Query Operators

- Here is what kind of geometry each geospatial operator uses:

| Query Type | Geometry Type |
|---|---|
| $*near* (GeoJSON point, 2dsphere index) | Spherical |
| $near (legacy coordinates, 2d index) | Flat |
| $*nearSphere* (GeoJSON point, 2dsphere index) | Spherical |
| $*nearSphere* (legacy coordinates, 2d index) | Spherical |
| $*geoWithin* : { $geometry: … } | Spherical |
| $*geoWithin* : { $box: … } | Flat |
| $*geoWithin* : { $polygon: … } | Flat |
| $*geoWithin* : { $center: … } | Flat |
| $*geoWithin* : { $centerSphere: … } | Spherical |
| $*geoIntersects* | Spherical |

Note: *geoNear* command and the $*geoNear* aggregation operator both operate in radians when using legacy coordinates, and meters when using GeoJSON points.

# MongoDB Geospatial Indexing

**Creating a 2dsphere Index**

- To create a *2dsphere* index, use the db.collection.createIndex() method and specify the string literal "2dsphere" as the index type:

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

  where the <location field> is a field whose value is either a GeoJSON object or a legacy coordinates pair.

- Unlike a compound *2d* index which can reference one location field and one other field, a compound *2dsphere* index can reference multiple location and non-location fields.

# MongoDB Geospatial Indexing

**Creating a 2dsphere Index**

- Consider a collection places with documents that store location data as GeoJSON Point in a field named loc:

```
db.places.insert(
  {
    loc: {type:"Point",coordinates: [ -73.97, 40.77 ] },
    name:"Central Park",
    category:"Parks”
  }
)
db.places.insert(
  {
    loc: {type:"Point",coordinates: [ -73.88, 40.78 ] },
    name:"La Guardia Airport",
    category:"Airport”
  }
)
```

# MongoDB Geospatial Indexing

**Creating a 2dsphere Index**

- The following operation creates a *2dsphere* index on the location field loc:

  ```
  db.places.createIndex( { loc : "2dsphere" } )
  ```

- The following operation creates a compound index where the first key loc is a *2dsphere* index key, and the remaining ones are non-geospatial index keys, specifically descending (-1) and ascending (1) keys respectively.

  ```
  db.places.createIndex( { loc : "2dsphere" , category : -1, name: 1 } )
  ```

- Unlike the *2d* index, a compound *2dsphere* index does not require the location field to be the first field indexed:

  ```
  db.places.createIndex( { category : 1 , loc : "2dsphere" } )
  ```

# MongoDB Geospatial Indexing

**Querying a 2dsphere Index: Bounded by Polygon**

- The $*geoWithin* operator queries for location data found within a GeoJSON polygon. Your location data must be stored in GeoJSON format.
- Use the following syntax:

```
db.<collection>.find(
    { <location field> :
        { $geoWithin :
            { $geometry :
                { type : "Polygon" ,
                  coordinates : [ <coordinates> ]
                }
            }
        }
    }
)
```

# MongoDB Geospatial Indexing

**Querying a 2dsphere Index: Bounded by Polygon**

- The following example selects all points and shapes that exist entirely within a GeoJSON polygon

```
db.places.find(
    { loc :
        { $geoWithin :
            { $geometry :
                { type : "Polygon" ,
                  coordinates : [
                      [ [ 0 , 0 ], [ 3 , 6 ], [ 6 , 1 ], [ 0 , 0 ] ]
                  ]
                }
            }
        }
    }
)
```

# MongoDB Geospatial Indexing

**Querying a 2dsphere Index: Intersection of GeoJSON Objects**

- The $geoIntersects operator queries for locations that intersect a specified GeoJSON object. A location intersects the object if the intersection is non-empty or have a shared edge.
- Use the following syntax:

```
db.<collection>.find(
    { <location field> :
        { $geoIntersects :
            { $geometry :
                { type : "<GeoJSON object type>" ,
                  coordinates : [ <coordinates> ]
                }
            }
        }
    }
)
```

# MongoDB Geospatial Indexing

**Querying a 2dsphere Index: Intersection of GeoJSON Objects**

- The following example uses $geoIntersects to select all indexed points and shapes that intersect with the polygon defined by the coordinates array.

```
db.places.find(
    { loc :
        { $geoIntersects :
            { $geometry :
                { type : "Polygon" ,
                  coordinates : [
                      [ [ 0 , 0 ], [ 3 , 6 ], [ 6 , 1 ], [ 0 , 0 ]  ]
                  ]
                }
            }
        }
    }
)
```

# MongoDB Geospatial Indexing

**Querying a 2dsphere Index: Proximity to GeoJSON Point**

- Proximity queries return the points closest to the defined point and sorts the results by distance (in meters). To query for proximity to a GeoJSON point, use the $near operator or geoNear command.
- The *$near* command has the following syntax:

```
db.<collection>.find(
    { <location field> :
        { $near :
            { $geometry :
                { type : "Point" ,
                  coordinates : [ <longitude>,  <latitude> ]
                } ,
              $maxDistance: <distance in meters>
            }
        }
    }
)
```

# MongoDB Geospatial Indexing

**Querying a 2dsphere Index: Proximity to GeoJSON Point**

- The $*geoNear* command has the following syntax:

```
db.runCommand(
    {  geoNear: <collection> :
       near : {
            type : ”Point” ,
            coordinates : [ <longitude>,  <latitude> ]
       } ,
       spherical: true
    }
)
```

# MongoDB Geospatial Indexing

**Querying a 2dsphere Index: Points Within Circle on Sphere**

- To select all grid coordinates in a “spherical cap” on a sphere, use $*geoWithin* with the $*centerSphere* operator. Specify an array that contains the grid coordinates of the circle’s center point, and the circle’s radius measured in radians
- Use the following syntax:

```
db.<collection>.find(
    { <location field> :
        { $geoWithin :
            { $centerSphere :
                [ [ <x>, <y> ], <radius> ]
            } ,
        }
    }
)
```

# MongoDB Geospatial Indexing

**Querying a 2dsphere Index: Points Within Circle on Sphere**

- The following example queries grid coordinates and returns all documents within a 10 km (~6 mi) radius of longitude 88 W and latitude 30 N. The example converts distance to radians by dividing by the approximate equatorial radius of the earth, 6371 km:
- Use the following syntax:

```
db.<collection>.find(
    { loc:
        { $geoWithin :
            { $centerSphere :
                [ [ -88, 30 ], 10 / 6371 ]
            } ,
        }
    }
)
```

# MongoDB Geospatial Indexing

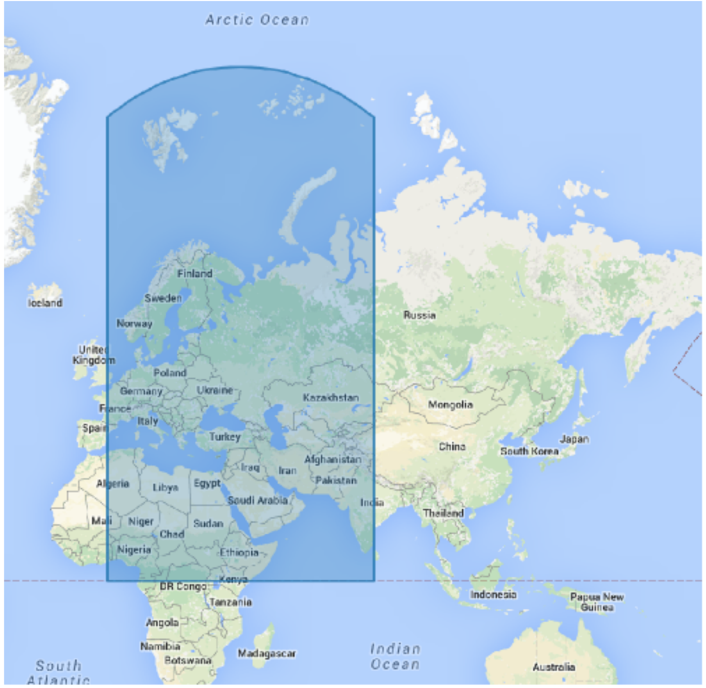
**Example Geospatial Application: Locating Restaurants**

- Suppose you are designing a mobile application to help users find restaurants in New York City.
- The application must:
  - Determine the user's current neighborhood using $*geoIntersects,*
  - Show the number of restaurants in that neighborhood using $*geoWithin,*
  - Find restaurants within a specified distance of the user using $*nearSphere.*
- We will use the *2dsphere* index to query the data on spherical geometry

# MongoDB Geospatial Indexing

**Distortion When Projecting Spherical Coordinates on a Plane**

- Spherical geometry appears distorted when visualized on a map due to the nature of projecting a three dimensional sphere, such as the earth, onto a flat plane.




Area covered by spherical square defined by the longitude latitude points (0,0), (80,0), (80,80), (0,80).

# MongoDB Geospatial Indexing

**Searching for Restaurants: Setup**

- Make sure the mongodb server is running .
- For this lecture, use *curl* or *wget* to download datasets from:
  - curl https://raw.githubusercontent.com/mongodb/docs-assets/geospatial/neighborhoods.json > ~/Downloads/neighborhoods.json
  - curl https://raw.githubusercontent.com/mongodb/docs-assets/geospatial/restaurants.json > ~/Downloads/restaurants.json
- Import the two datasets using *mongoimport*:

  mongoimport *~/Downloads/restaurants.json* -c restaurants

  mongoimport *~/Downloads/neighborhoods.json* -c neighborhoods

# MongoDB Geospatial Indexing

**Searching for Restaurants: Indexes**

- The *geoNear* command requires a geospatial index, and almost always improves performance of $*geoWithin* and $*geoIntersects* queries.
- Because this data is geographical, create a *2dsphere* index on each collection using the mongo shell:

```
db.restaurants.createIndex({ location: "2dsphere" })
db.neighborhoods.createIndex({ geometry: "2dsphere" })
```

# MongoDB Geospatial Indexing

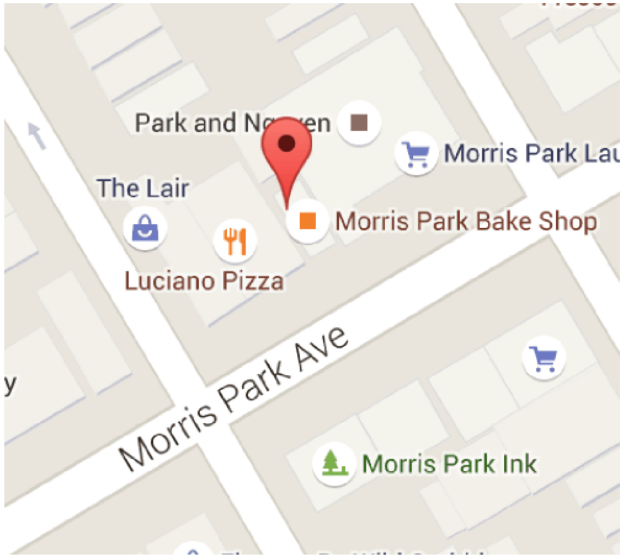
## Exploring the Data

- Inspect an entry in the newly-created restaurants collection from within the mongo shell:

```
db.restaurants.findOne()
```

- This query returns:

```
{
    location: {
        type: "Point",
        coordinates: [-73.856077, 40.848447]
    },
    name: "Morris Park Bake Shop"
}
```




- The geometry data in the location field follows the doc:GeoJSON format </reference/geojson>.

# MongoDB Geospatial Indexing

**Exploring the Data**

- Inspect an entry in the newly-created neighborhoods collection:

```
db.neighborhoods.findOne()
```


- This query returns:

```
{
    geometry: {
        type: "Polygon",
        coordinates: [[
            [ -73.99, 40.75 ],
            ...
            [ -73.98, 40.76 ],
            [ -73.99, 40.75 ]
        ]]
    },
    name: "Hell's Kitchen”
}
```

# MongoDB Geospatial Indexing

**Find the Current Neighborhood**

- Suppose that your cell phone show the location -73.93414657 longitude and 40.82302903 latitude.
- To find the current neighborhood, you specify a point using the special $*geometry* field in GeoJSON format:

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

# MongoDB Geospatial Indexing

**Find the Current Neighborhood**

- The query returns the following result:


```
{
   "_id" : ObjectId("55cb9c666c522cafdb053a68"),
   "geometry" : {
      "type" : "Polygon",
      "coordinates" : [
         [
            [
               -73.93383000695911,
               40.81949109558767
            ],
            ...
         ]
      ]
   },
   "name" : "Central Harlem North-Polo Grounds”
}
```

# MongoDB Geospatial Indexing

**Find Restaurants in the Neighborhood**

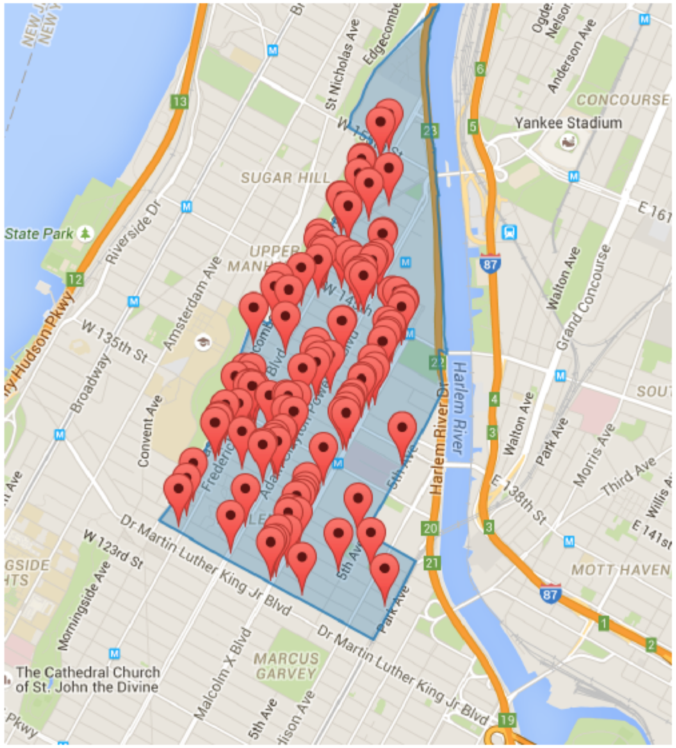
- The following command finds the neighborhood containing the user, and then counts the restaurants within that neighborhood:

```
var neighborhood = db.neighborhoods.findOne(
    {
        geometry: {
            $geoIntersects: {
                $geometry: { type: "Point",  coordinates: [ -73.93414657, 40.82302903 ] }
            }
        }
    }
)
db.restaurants.find(
    {
        location: {
            $geoWithin: {$geometry: neighborhood.geometry }
        }
    }
).count()
```

# MongoDB Geospatial Indexing

**Find Restaurants in the Neighborhood**

- The query shows that there are 127 restaurants in the requested neighborhood, as shown on this map:



# MongoDB Geospatial Indexing

**Find  Restaurants Within a Distance**

- Now, we will find restaurants within a specified distance of our location.
- You can use either
  - $geoWithin with $centerSphere to return results in unsorted order
  - nearSphere with $maxDistance if you need results sorted by distance.

# MongoDB Geospatial Indexing

**Find Restaurants Within a Distance: $*geoWithin (unsorted)***

- To find restaurants within a circular region, use $*geoWithin* with $*centerSphere*.
- $*centerSphere* is a MongoDB-specific syntax to denote a circular region by specifying the center and the radius in radians.
- $*geoWithin* does not return the documents in any specific order, so it may show the user the furthest documents first.

# MongoDB Geospatial Indexing

**Find  Restaurants Within a Distance: *$geoWithin (unsorted)***

- This query finds all restaurants within 10 km (~6 miles) of the user

```
db.restaurants.find(
    {
        location:   {
            $geoWithin:  { $centerSphere: [ [ -73.93414657, 40.82302903 ], 10 / 6371 ]  }
        }
    }
)
```

- *$centerSphere*'s second argument accepts the radius in radians, so you must divide it by the radius of the earth in kilometers: 6371

# MongoDB Geospatial Indexing

**Find Restaurants Within a Distance: $*nearSphere (sorted)***

- You may also use $*nearSphere* and specify a $*maxDistance* term in meters.
- This will return all restaurants within 10 km (~6 miles) of the user in sorted order from nearest to farthest:

```
db.restaurants.find(
   {
      location: {
         $nearSphere: {
            $geometry: {
               type: "Point",
               coordinates: [ -73.93414657, 40.82302903 ]
            },
            $maxDistance: 10000
         }
      }
   }
)
```