CLASS 4: PROJECTION & LIMIT

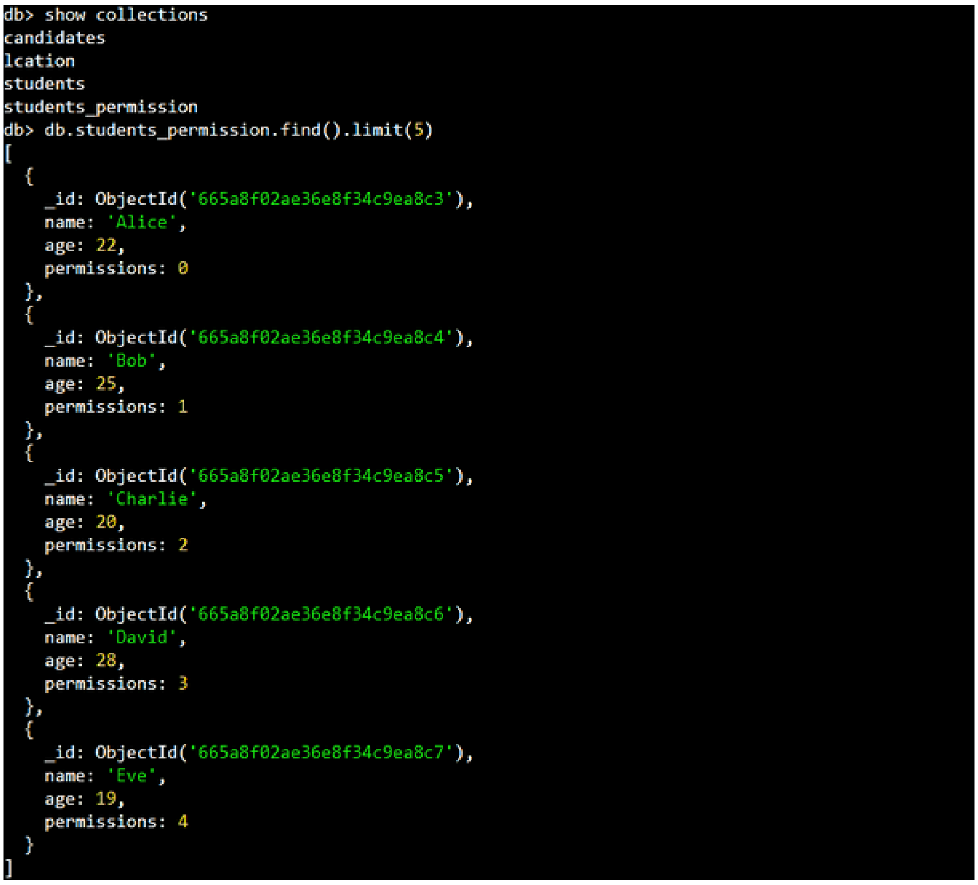
LIMIT

In MongoDB, limits are used to control the number of documents returned by a query. This is especially useful for managing large datasets and retrieving a manageable subset of documents. The primary method used for limiting the number of documents is the **limit()** method. Here's a detailed explanation of limits in MongoDB with examples.

Syntax
db.collection.find(query).limit(number)

- collection: The collection to query.
- query: The criteria for selecting documents.
- number: The maximum number of documents to return.

Basic Usage of limit():
Retrieve the First 5 Students



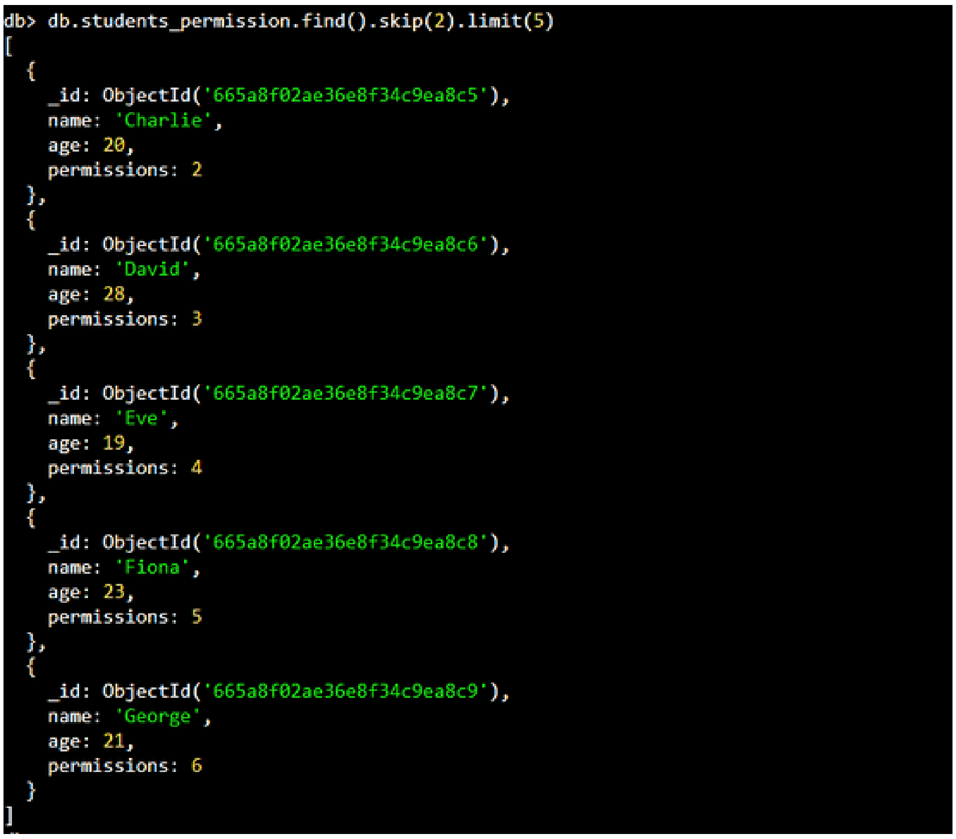
```
db> show collections
candidates
lcation
students
students_permission
db> db.students_permission.find().limit(5)
[
  {
    _id: ObjectId('665a8f02ae36e8f34c9ea8c3'),
    name: 'Alice',
    age: 22,
    permissions: 0
  },
  {
    _id: ObjectId('665a8f02ae36e8f34c9ea8c4'),
    name: 'Bob',
    age: 25,
    permissions: 1
  },
  {
    _id: ObjectId('665a8f02ae36e8f34c9ea8c5'),
    name: 'Charlie',
    age: 20,
    permissions: 2
  },
  {
    _id: ObjectId('665a8f02ae36e8f34c9ea8c6'),
    name: 'David',
    age: 28,
    permissions: 3
  },
  {
    _id: ObjectId('665a8f02ae36e8f34c9ea8c7'),
    name: 'Eve',
    age: 19,
    permissions: 4
  }
]
```

Combining limit() with Other Query Modifiers:

Skip and Limit:

The **skip()** method is used in conjunction with **limit()** to implement pagination.

Example: Skip the First 2 Students and Retrieve the Next 5

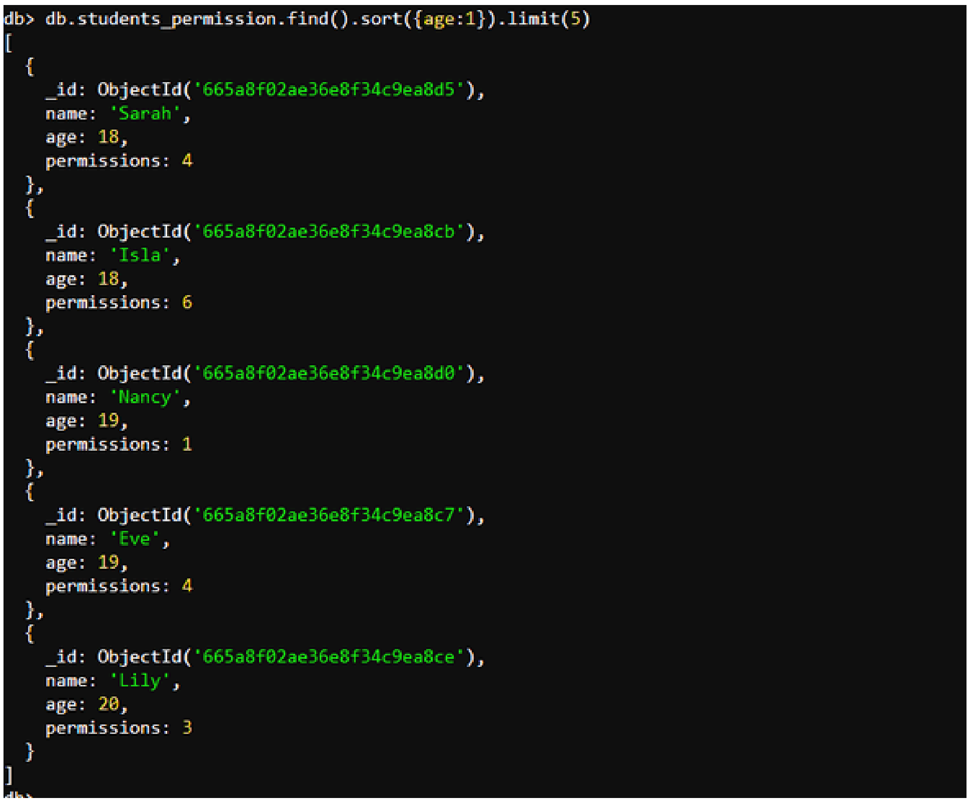


```
db> db.students_permission.find().skip(2).limit(5)
[
  {
    _id: ObjectId('665a8f02ae36e8f34c9ea8c5'),
    name: 'Charlie',
    age: 20,
    permissions: 2
  },
  {
    _id: ObjectId('665a8f02ae36e8f34c9ea8c6'),
    name: 'David',
    age: 28,
    permissions: 3
  },
  {
    _id: ObjectId('665a8f02ae36e8f34c9ea8c7'),
    name: 'Eve',
    age: 19,
    permissions: 4
  },
  {
    _id: ObjectId('665a8f02ae36e8f34c9ea8c8'),
    name: 'Fiona',
    age: 23,
    permissions: 5
  },
  {
    _id: ObjectId('665a8f02ae36e8f34c9ea8c9'),
    name: 'George',
    age: 21,
    permissions: 6
  }
]
```

Sort and Limit:

The **sort()** method can be combined with **limit()** to retrieve a limited number of documents in a specific order.

Example: Retrieve the First 5 Students Sorted by Age

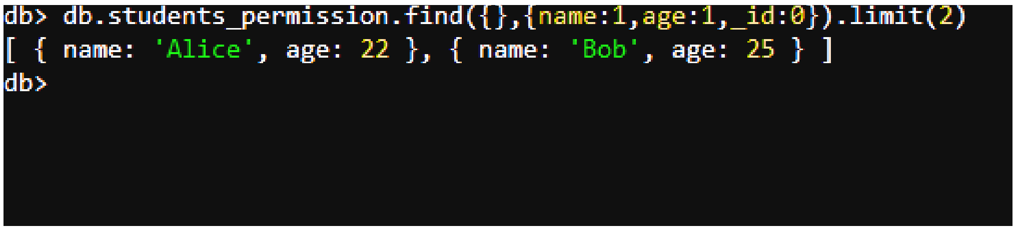


```
db> db.students_permission.find().sort({age:1}).limit(5)
[
  {
    _id: ObjectId('665a8f02ae36e8f34c9ea8d5'),
    name: 'Sarah',
    age: 18,
    permissions: 4
  },
  {
    _id: ObjectId('665a8f02ae36e8f34c9ea8cb'),
    name: 'Isla',
    age: 18,
    permissions: 6
  },
  {
    _id: ObjectId('665a8f02ae36e8f34c9ea8d0'),
    name: 'Nancy',
    age: 19,
    permissions: 1
  },
  {
    _id: ObjectId('665a8f02ae36e8f34c9ea8c7'),
    name: 'Eve',
    age: 19,
    permissions: 4
  },
  {
    _id: ObjectId('665a8f02ae36e8f34c9ea8ce'),
    name: 'Lily',
    age: 20,
    permissions: 3
  }
]
```

Using limit() with Projection

Projection can be used to specify which fields to return in the result set.

Example: Retrieve the First 2 Students with Only Names and Ages



```
db> db.students_permission.find({},{name:1,age:1,_id:0}).limit(2)
[ { name: 'Alice', age: 22 }, { name: 'Bob', age: 25 } ]
db>
```

GEOSPATIAL:

In MongoDB, geospatial data can be stored and queried using special geospatial indexes and data types. The example document you've provided already includes a geospatial field using the GeoJSON format, which is a standard format for encoding various geographic data structures.Given the extended dataset, let's incorporate these additional documents into the locations collection and perform geospatial queries.
Example: Find all locations near the coordinates [-74.00597, 40.71278] within a 1000-meter radius.

Key Concepts:

- **GeoJSON:** A format for encoding a variety of geographic data structures using JavaScript Object Notation (JSON).
- **Geospatial Indexes:** Special indexes used to optimize geospatial queries.
- 2dsphere Index: Supports queries for data stored as GeoJSON objects and for legacy coordinate pairs; for spherical (Earth-like) geometry.

Data Types:

- **Point**: Represents a single geographic point.
- **LineString**: Represents a series of connected points.
- **Polygon:** Represents an enclosed area defined by three or more points.

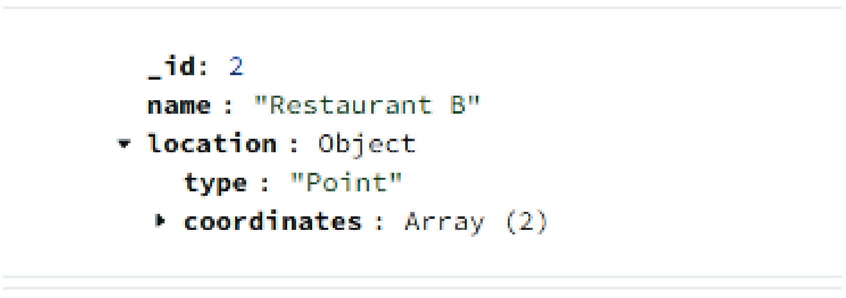
Creating a Geospatial Index in MongoDB:

Creating a 2dsphere index on the location field in MongoDB allows you to efficiently perform geospatial queries on your data. This index type supports queries for data stored as GeoJSON objects and for legacy coordinate pairs. Here's how to create and use a 2dsphere index.

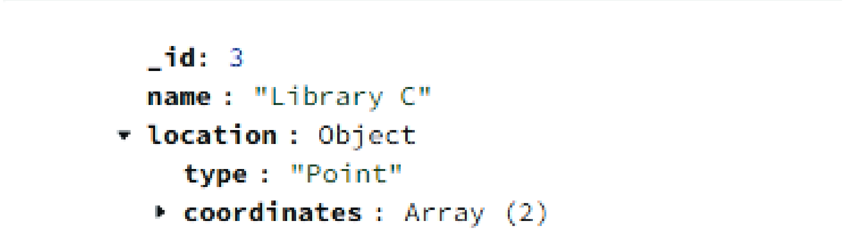
Why Use a 2dsphere Index?

- Efficient Querying: It enables efficient querying of geospatial data.
- Support for GeoJSON: It supports the GeoJSON format, which is a standard for representing geographical features.

Steps to Create a 2dsphere Index:
1.Ensure Your Data is in GeoJSON Format:Your documents should contain geospatial data in the GeoJSON format.
Example document:



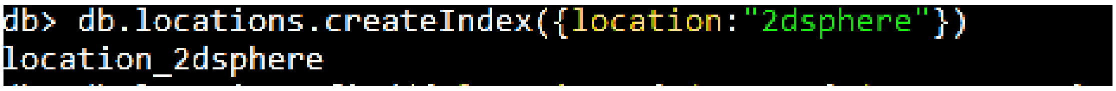
```
_id: 2
name : "Restaurant B"
▾ location : Object
    type : "Point"
  ▸ coordinates : Array (2)
```



```
_id: 3
name : "Library C"
▾ location : Object
    type : "Point"
  ▸ coordinates : Array (2)
```

2.Create the 2dsphere Index:

.Use the createIndex method to create the 2dsphere index on the location field.



```
db> db.locations.createIndex({location:"2dsphere"})
location_2dsphere
```

How It Works:

When you create a 2dsphere index on a field, MongoDB indexes the field's geospatial data. This index allows MongoDB to quickly locate documents based on their geographic location.

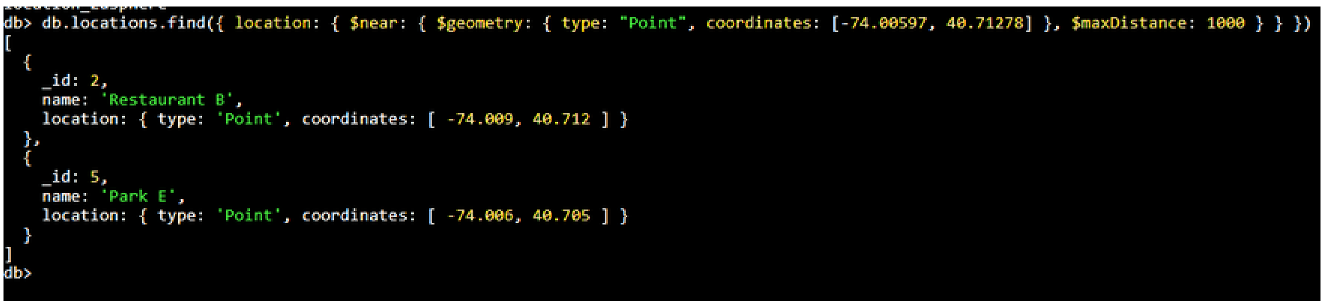
With the 2dsphere index in place, you can perform various geospatial queries.

Find Near a Point:

The $near operator in MongoDB is used to find documents with geospatial data that are close to a specified point. When combined with a 2dsphere index, this operator can efficiently locate documents based on their proximity to a given geographic coordinate.

Syntax:

```
db.collection.find({
    location: {
        $near: {
            $geometry: {
                type: "Point",
                coordinates: [longitude, latitude]
            },
            $maxDistance: distance_in_meters  // Optional: Maximum distance in meters from the
point
        }
    }
})
```



```
db> db.locations.find({ location: { $near: { $geometry: { type: "Point", coordinates: [-74.00597, 40.71278] }, $maxDistance: 1000 } } })
[
  {
    _id: 2,
    name: 'Restaurant B',
    location: { type: 'Point', coordinates: [ -74.009, 40.712 ] }
  },
  {
    _id: 5,
    name: 'Park E',
    location: { type: 'Point', coordinates: [ -74.006, 40.705 ] }
  }
]
db>
```

Explanation

1. location: This is the field that contains the geospatial data.
2. $near: This operator specifies that the query should find documents close to the given point.
3. $geometry: This sub-operator specifies the type and coordinates of the reference point.
    - type: Indicates the type of GeoJSON object (usually "Point" for a single point).
    - coordinates: Specifies the longitude and latitude of the reference point.
4. $maxDistance: An optional parameter that limits the search to documents within the specified distance (in meters) from the point.

By using the **$near** operator, you can efficiently perform geospatial searches to find the nearest locations based on your requirements.

Within a Radius:

Finding documents within a specific radius in MongoDB involves using the $geoWithin operator along with the $centerSphere operator. This is particularly useful for queries that need to locate all points within a circular area defined by a center point and a radius.

How the $geoWithin Operator Works

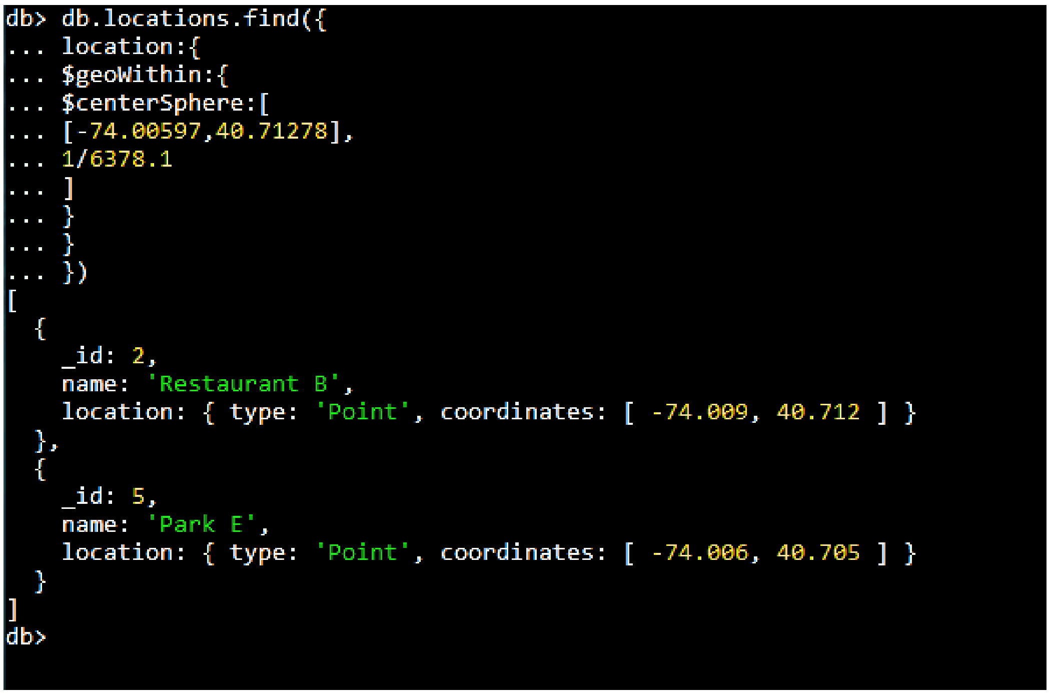
- **Purpose:** The $geoWithin operator is used to find documents that are within a specified geometry.
- **Usage:** Combined with $centerSphere, it allows you to define a circular area in terms of its center coordinates and radius.

Syntax:

```
db.collection.find({
   location: {
      $geoWithin: {
         $centerSphere: [ [ <longitude>, <latitude> ], <radius in radians> ]
      }
   }
})
```

Example Scenario

Consider a collection named locations ,To find locations within a 1 km radius from the point [-74.00597, 40.71278]:



```
db> db.locations.find({
... location:{
... $geoWithin:{
... $centerSphere:[
... [-74.00597,40.71278],
... 1/6378.1
... ]
... }
... }
... })
[
  {
    _id: 2,
    name: 'Restaurant B',
    location: { type: 'Point', coordinates: [ -74.009, 40.712 ] }
  },
  {
    _id: 5,
    name: 'Park E',
    location: { type: 'Point', coordinates: [ -74.006, 40.705 ] }
  }
]
db>
```

Explanation

1. location: This is the field that contains the geospatial data.
2. $geoWithin: This operator specifies that the query should find documents within a certain geometry.
3. $centerSphere: This sub-operator defines the circular area in terms of its center and radius.
   - [<longitude>, <latitude>]: The center coordinates of the circle.
   - <radius in radians>: The radius of the circle. Since MongoDB expects the radius in radians, you need to convert kilometers to radians. The Earth's radius is approximately 6378.1 kilometers. Thus, 1 km is approximately 1 / 6378.1 radians.

By using the **$geoWithin** and **$centerSphere** operators, you can efficiently query for documents within a circular area, making it ideal for tasks like finding nearby locations within a certain distance.

Within a Polygon:

MongoDB allows you to find documents whose geospatial data falls within a specified polygon using the $geoWithin operator combined with the $geometry sub-operator. This is useful for applications that need to determine whether points fall within complex boundaries such as city limits, neighborhoods, or custom-defined areas.

How the $geoWithin Operator Works

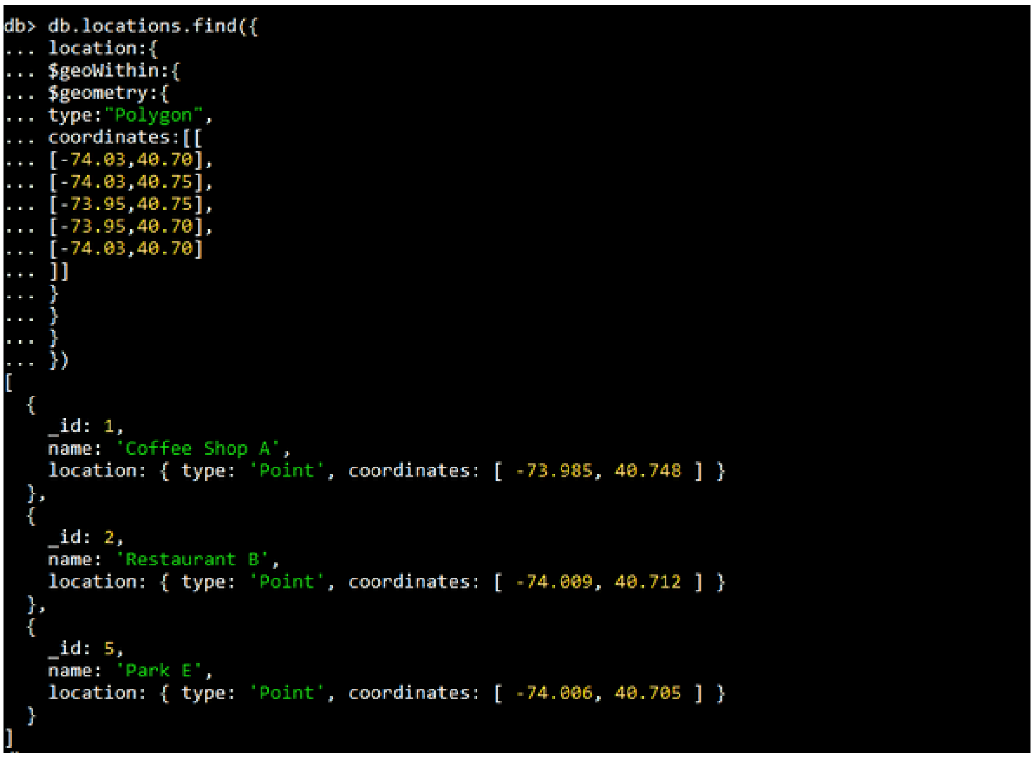
- Purpose: The $geoWithin operator finds documents that are located within a specified geometry, such as a polygon.
- Usage: By using $geometry with a GeoJSON Polygon, you can define complex shapes to query documents.

Syntax:

```
db.collection.find({
    location: {
        $geoWithin: {
            $geometry: {
                type: "Polygon",
                coordinates: [
                    [
                        [longitude1, latitude1],
                        [longitude2, latitude2],
                        [longitude3, latitude3],
                        ...,
                        [longitudeN, latitudeN],
                              [longitude1, latitude1]  // The polygon must be closed, so the first and last
points are the same
                    ]
                ]
            }
        }
    }
})
```

Example Scenario

Consider a collection named locations To find locations within a polygon defined by a set of coordinates:



```
db> db.locations.find({
... location:{
... $geoWithin:{
... $geometry:{
... type:"Polygon",
... coordinates:[[
... [-74.03,40.70],
... [-74.03,40.75],
... [-73.95,40.75],
... [-73.95,40.70],
... [-74.03,40.70]
... ]]
... }
... }
... }
... })
[
  {
    _id: 1,
    name: 'Coffee Shop A',
    location: { type: 'Point', coordinates: [ -73.985, 40.748 ] }
  },
  {
    _id: 2,
    name: 'Restaurant B',
    location: { type: 'Point', coordinates: [ -74.009, 40.712 ] }
  },
  {
    _id: 5,
    name: 'Park E',
    location: { type: 'Point', coordinates: [ -74.006, 40.705 ] }
  }
]
```

Explanation

1. location: The field that contains the geospatial data.
2. $geoWithin: This operator specifies that the query should find documents within the given geometry.
3. $geometry: This sub-operator specifies the type and coordinates of the geometry.
    - type: Indicates the type of GeoJSON object, which is "Polygon" in this case.
    - coordinates: An array of arrays of longitude and latitude pairs defining the vertices of the polygon. The first and last coordinate pairs must be the same to close the polygon.

By using the **$geoWithin** operator with a polygon, you can efficiently query for documents within complex geographical boundaries. This is ideal for applications that need to filter data based on specific geographic areas.