**PART A:**

* Creating a Geographic Information System (GIS) Analysis using MongoDB  
   Database created as “GIS”
* Collection name: “geodata”

**1.Retrieve Locations of specific features:**

Step1 : Insert 6 location coordinate in collection named('geodata')

Location 1:

db.geodata.insertOne({

'name':'Dunkin',

'location':{

'type':'Point',

'coordinates':[73.7033602,40.752222]

}

});

Location 2:

db.geodata.insertOne({

'name':'Venom',

'location':{

'type':'Point',

'coordinates':[73.7217817,40.7253564]

}

});

Location 3:

db.geodata.insertOne({

'name':'Pace University',

'location':{

'type':'Point',

'coordinates':[74.0051758,40.7112562]

}

});

Location 4:

db.geodata.insertOne({

'name':'Journal square',

'location':{

'type':'Point',

'coordinates':[74.0631544,40.7345715]

}

});

Location 5:

db.geodata.insertOne({

'name':'Time square',

'location':{

'type':'Point',

'coordinates':[73.9881175,40.7579747]

}

});

Location 6:

db.geodata.insertOne({

'name':'Long Island University',

'location':{

'type':'Point',

'coordinates':[73.9992087,40.8700085]

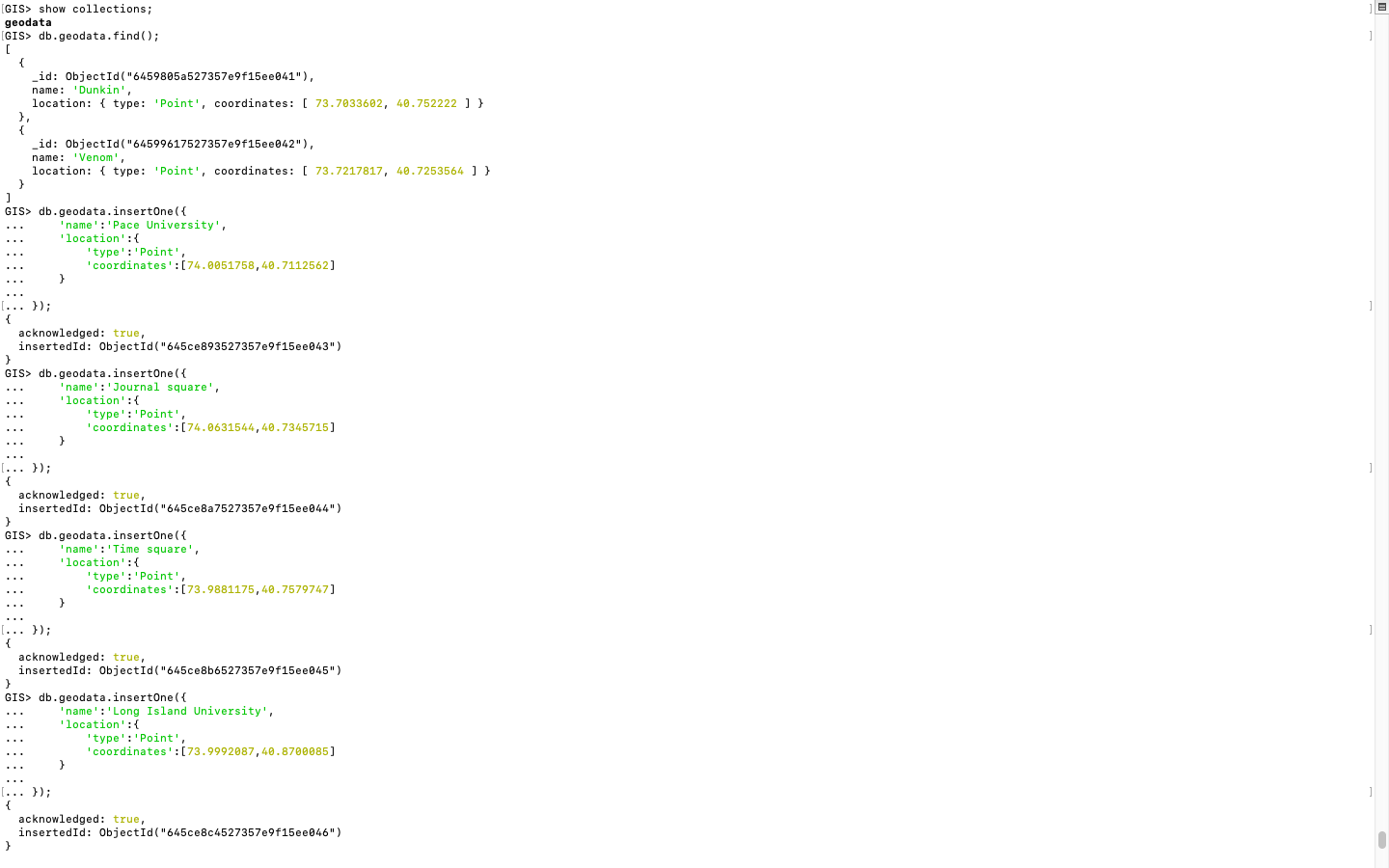
}

});

Step 2: Create an index for geodata

db.geodata.createIndex({location:'2dsphere'});

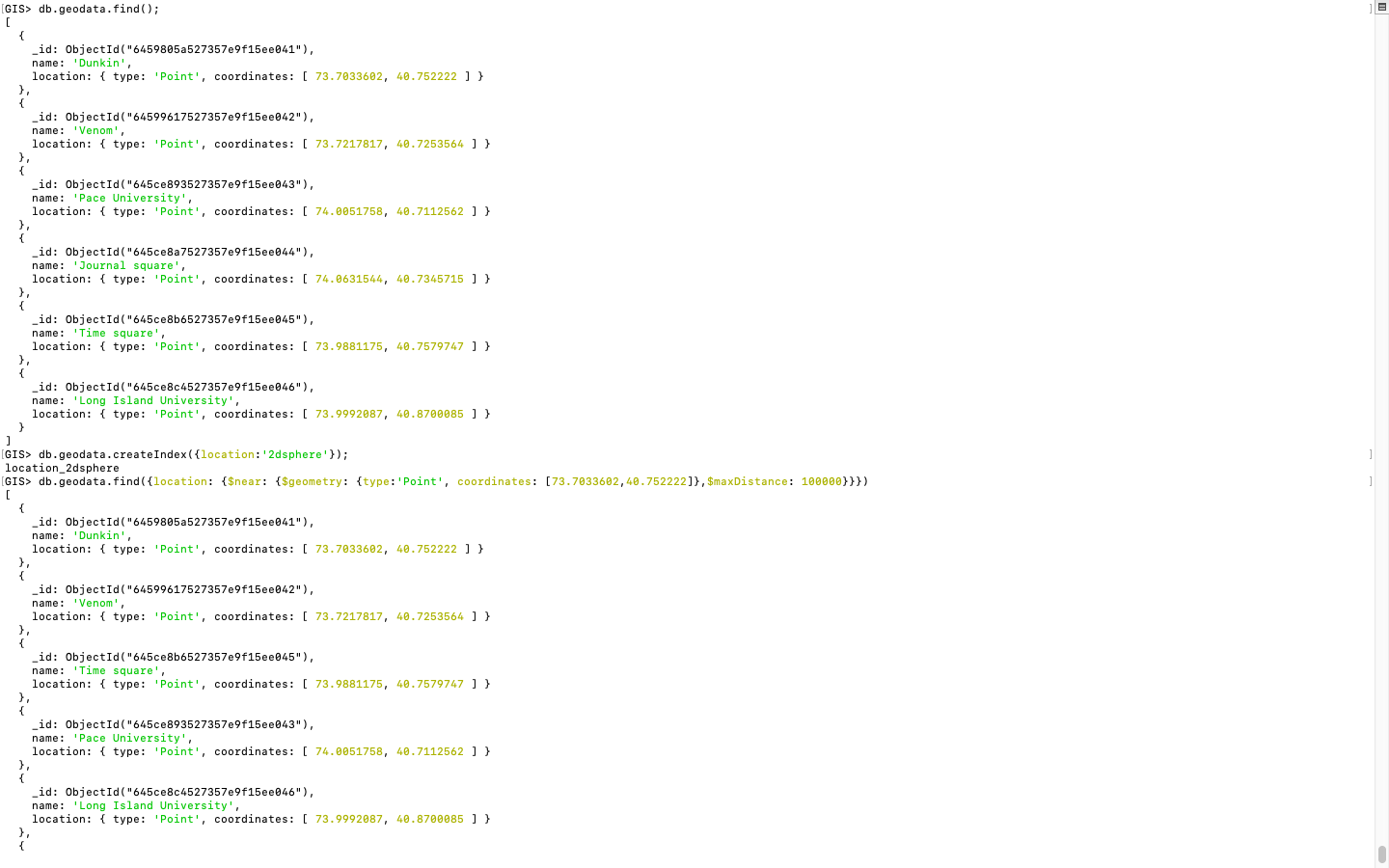
O/p:



Step 3: Once the index is created, you can query the collection to retrieve locations based on your specific criteria using $near

db.geodata.find({location: {$near: {$geometry: {type:'Point', coordinates: [73.7033602,40.752222]},$maxDistance: 100000}}})

O/p:



2.**Calculate Distance between points**

To calculate the distance between two specific points, limiting the stage result to 6 documents

db.geodata.aggregate([

{

$geoNear: {

near: {

type: "Point",

coordinates: [74.0051758,40.7112562],

coordinates: [73.9992087,40.8700085]

},

distanceField: "distance",

spherical: true

}

},

{ $limit: 6 }, //

$addFields: {

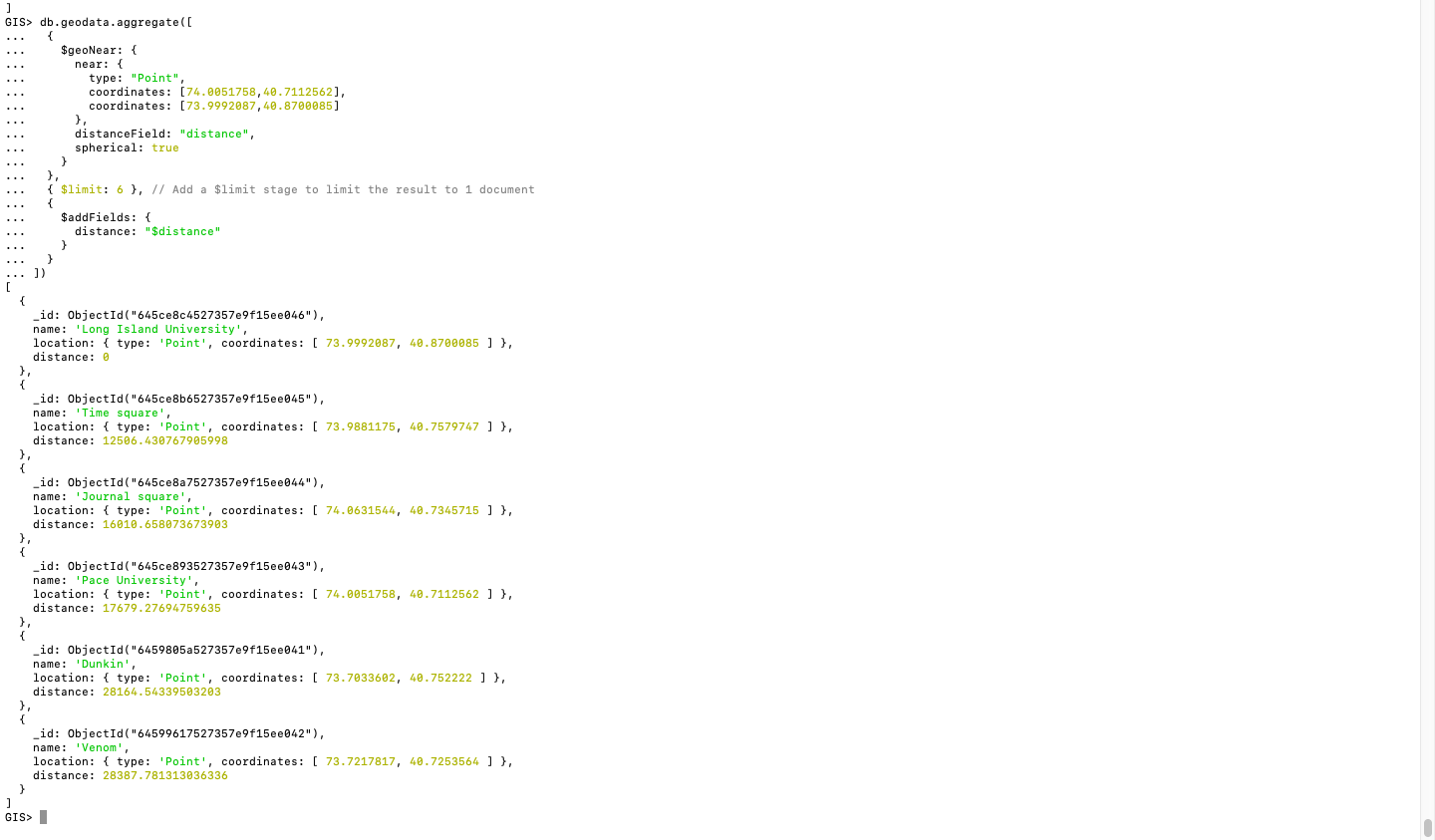
distance: "$distance"

}

}

])

O/p:



3.**Calculate Areas of Interest (specific to each group)**

Use the $geoWithin operator to find documents that fall within a specific area of interest.

db.geodata.find({

location: {

$geoWithin: {

$geometry: {

type: 'Polygon',

coordinates: [

[

[73.7033602,40.752222],

[74.0051758,40.7112562],

[74.0631544,40.7345715],

[73.9881175,40.7579747],

[73.7033602,40.752222]

]

]

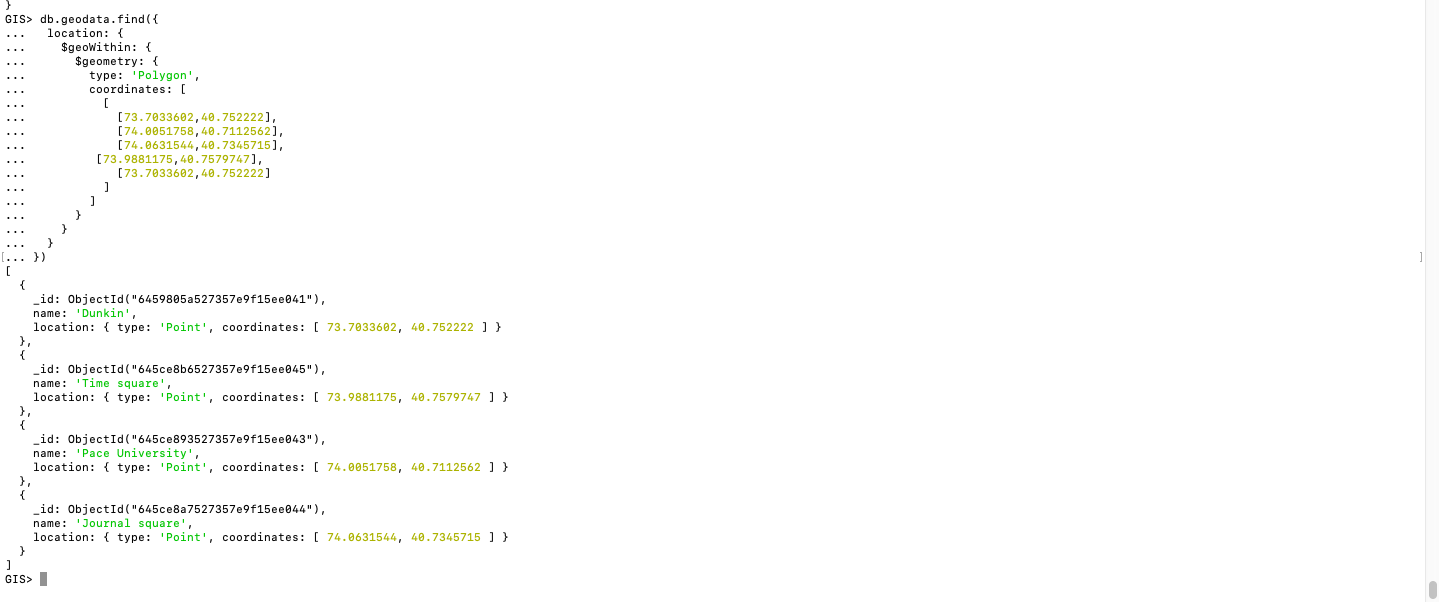
}

}

}

})

O/p:



To calculate areas of interest based on multiple polygons or complex geometries, you can use the GeoJSON MultiPolygon format.

db.geodata.find({

location: {

$geoWithin: {

$geometry: {

type: "MultiPolygon",

coordinates: [

[

[

[73.7033602,40.752222],

[74.0051758,40.7112562],

[74.0631544,40.7345715],

[73.7033602,40.752222]

]

],

[

[

[73.9881175,40.7579747],

[73.9992087,40.8700085],

[73.7217817,40.7253564],

[73.9881175,40.7579747]

]

]

]

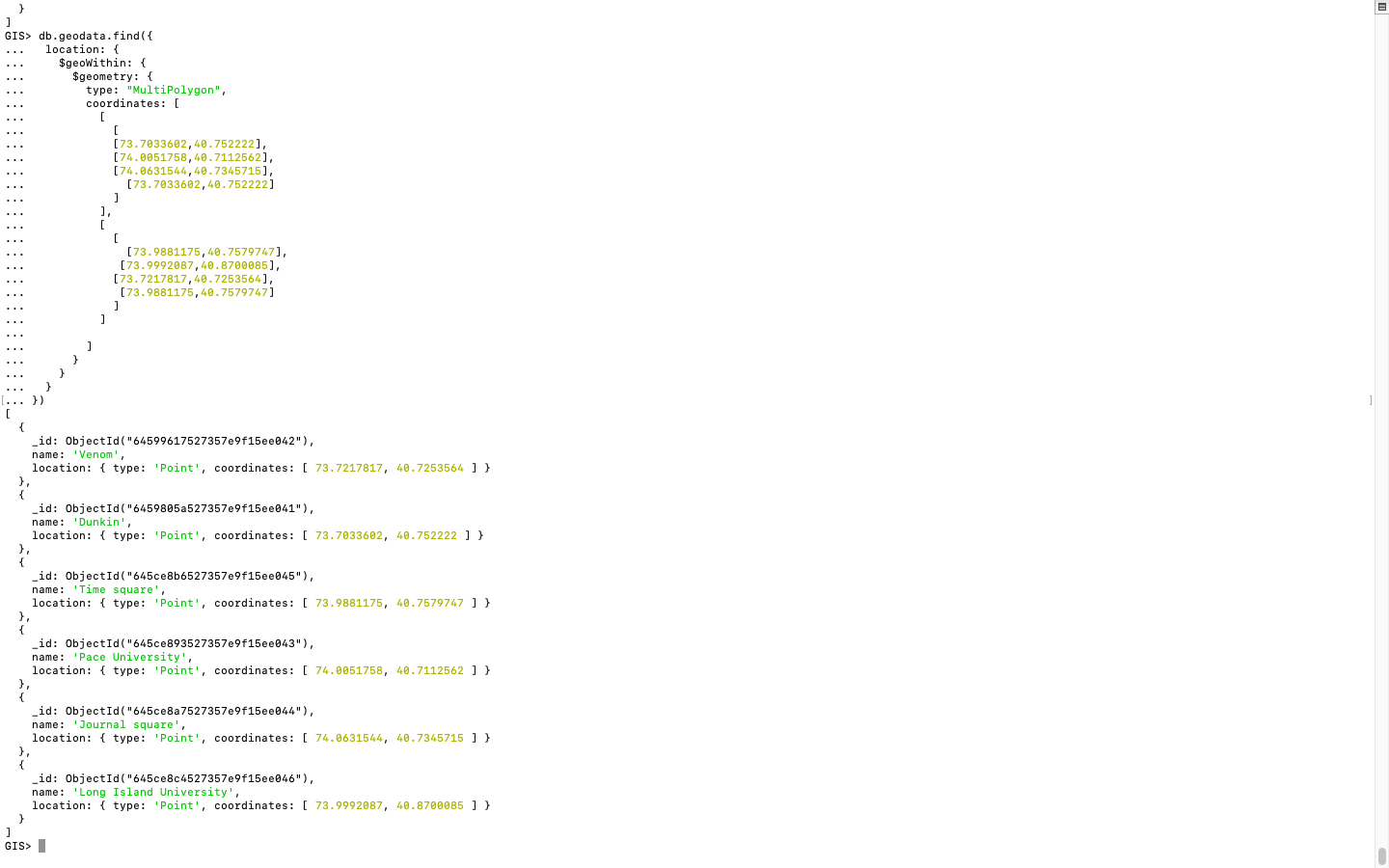
}

}

}

})

O/p:



4.**Analyze the queries**

Step 1: For retrieving locations of specific features, you can use the explain() method with the find()

db.geodata.find({

location: {

$geoWithin: {

$geometry: {

type: 'Polygon',

coordinates: [

[

[73.7033602,40.752222],

[74.0051758,40.7112562],

[74.0631544,40.7345715],

[73.9881175,40.7579747],

[73.7033602,40.752222]

]

]

}

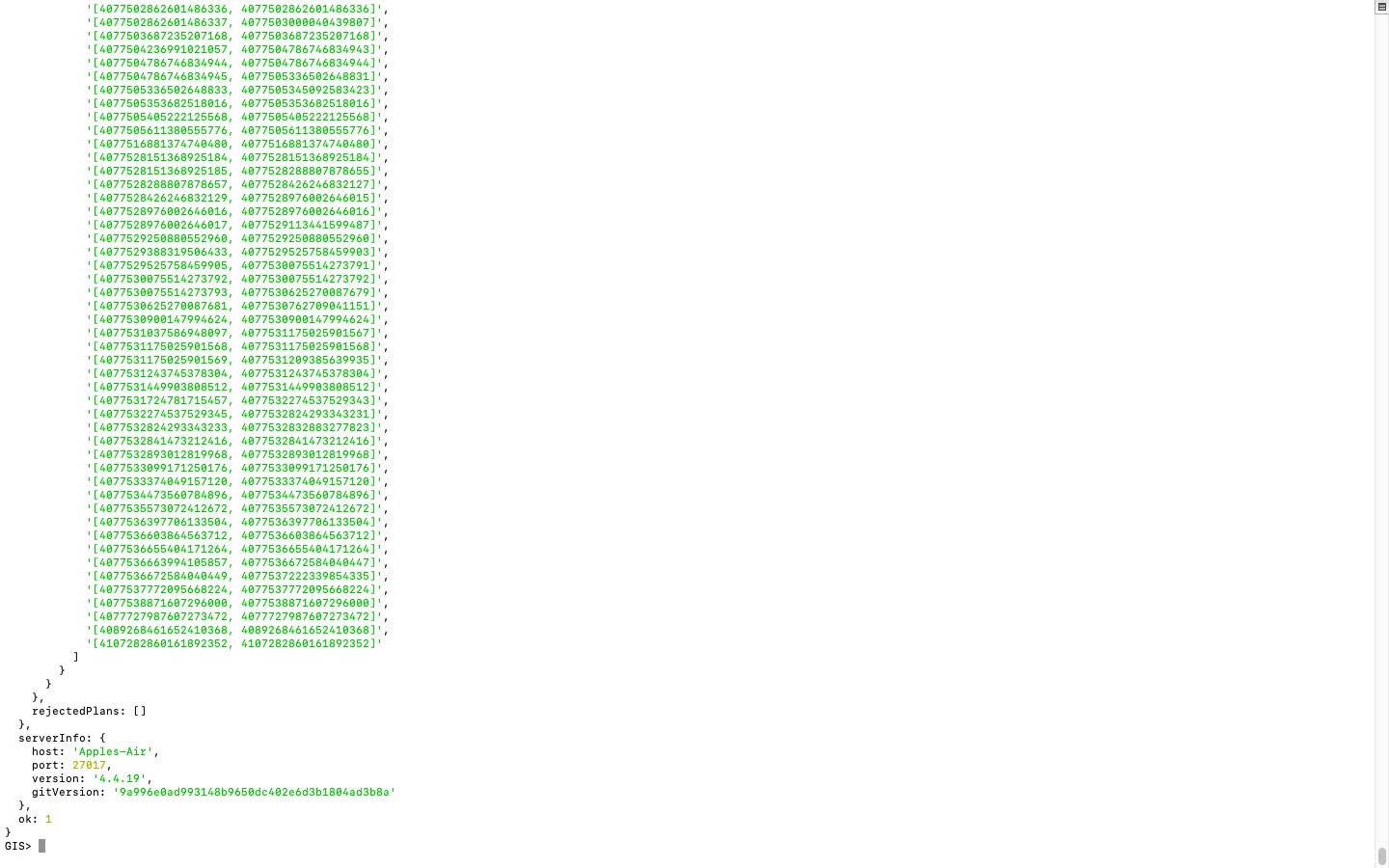
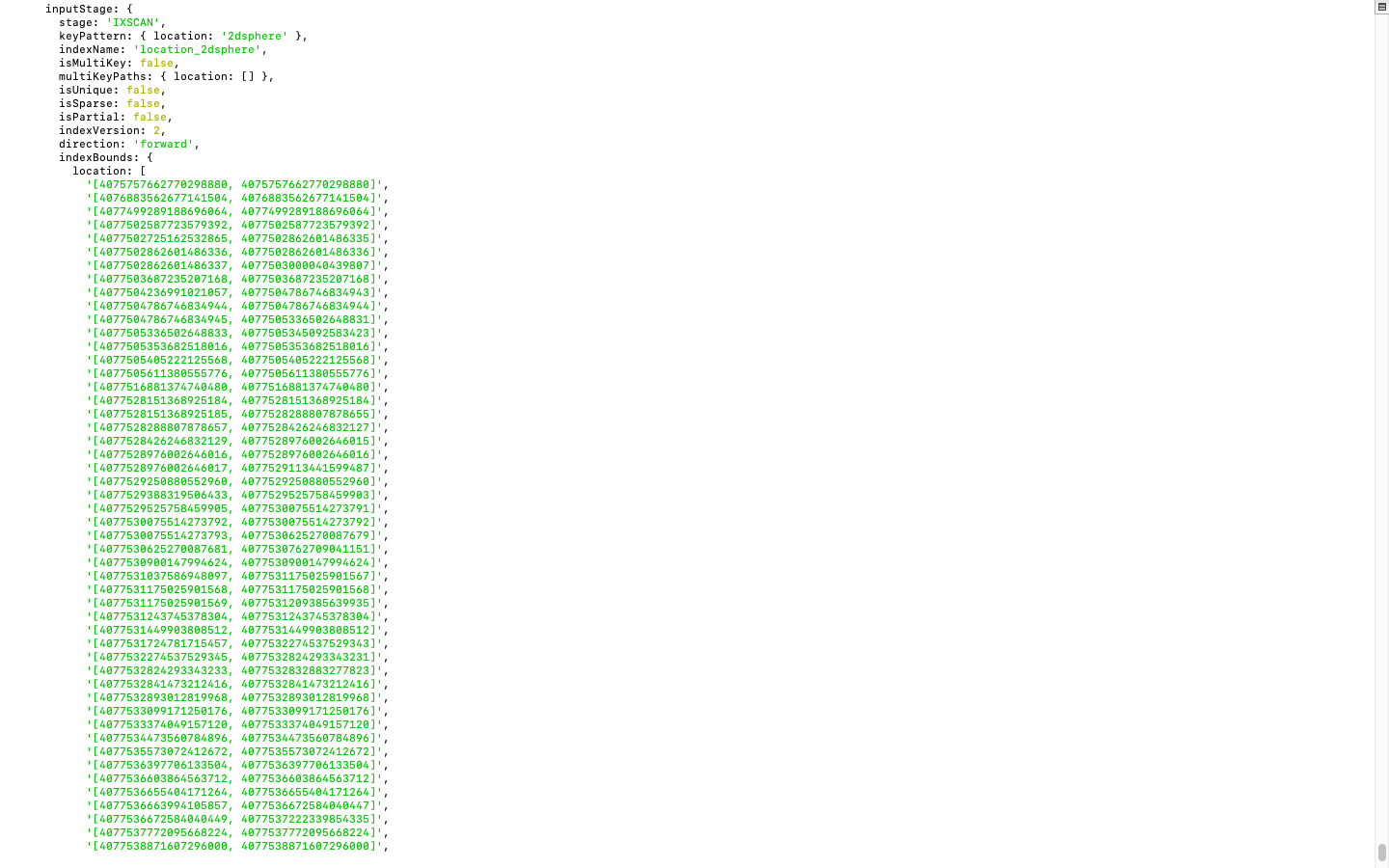
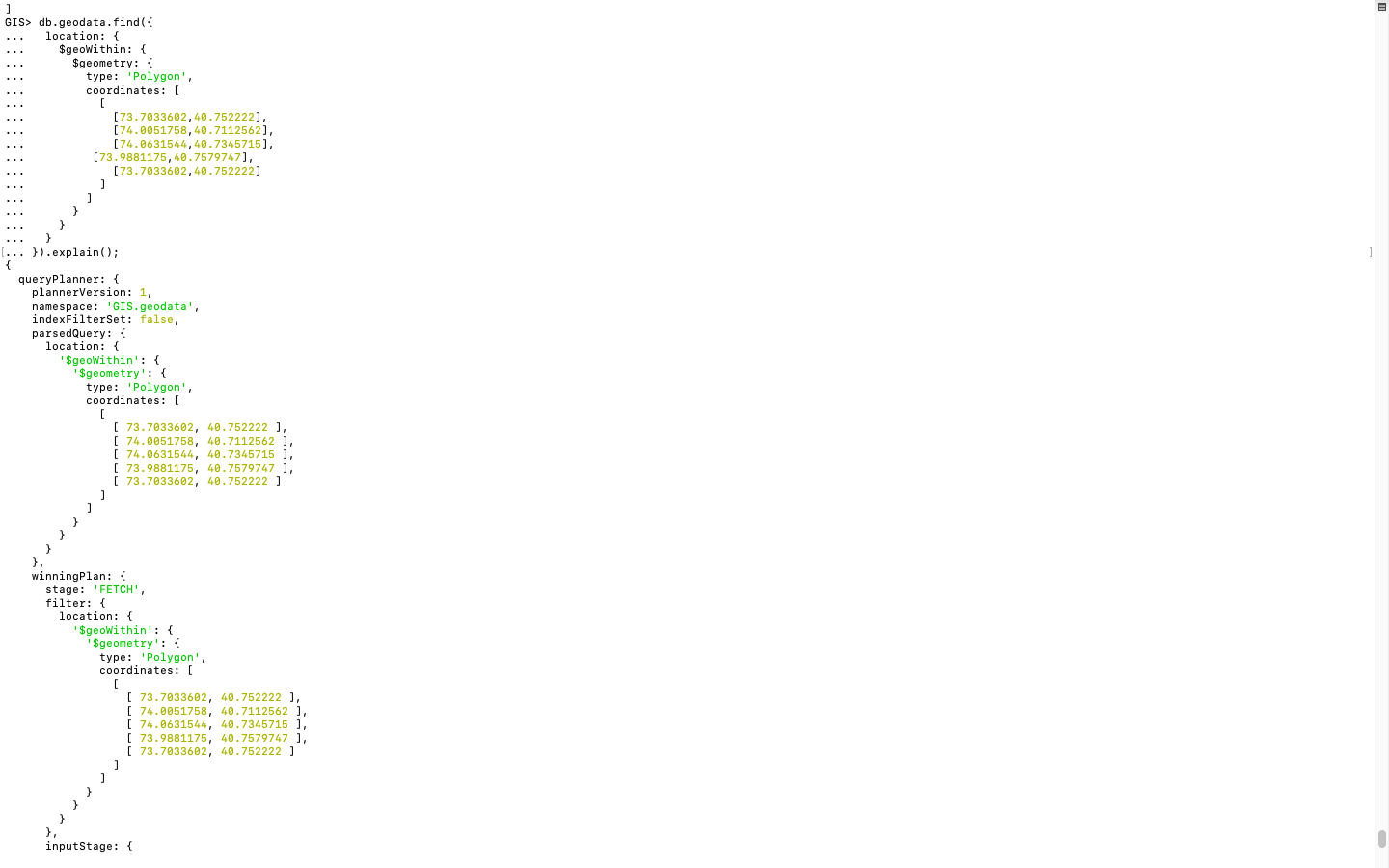
}

}

}).explain();

O/p:

Insert o/p 5i 5ii 5iii here



Step 2: For calculating distances between points, you can use the explain()

db.geodata.aggregate([

{

$geoNear: {

near: {

type: "Point",

coordinates: [74.0051758,40.7112562],

coordinates: [73.9992087,40.8700085]

},

distanceField: "distance",

spherical: true

}

},

{ $limit: 6 }, // Add a $limit stage to limit the result to 1 document

{

$addFields: {

distance: "$distance"

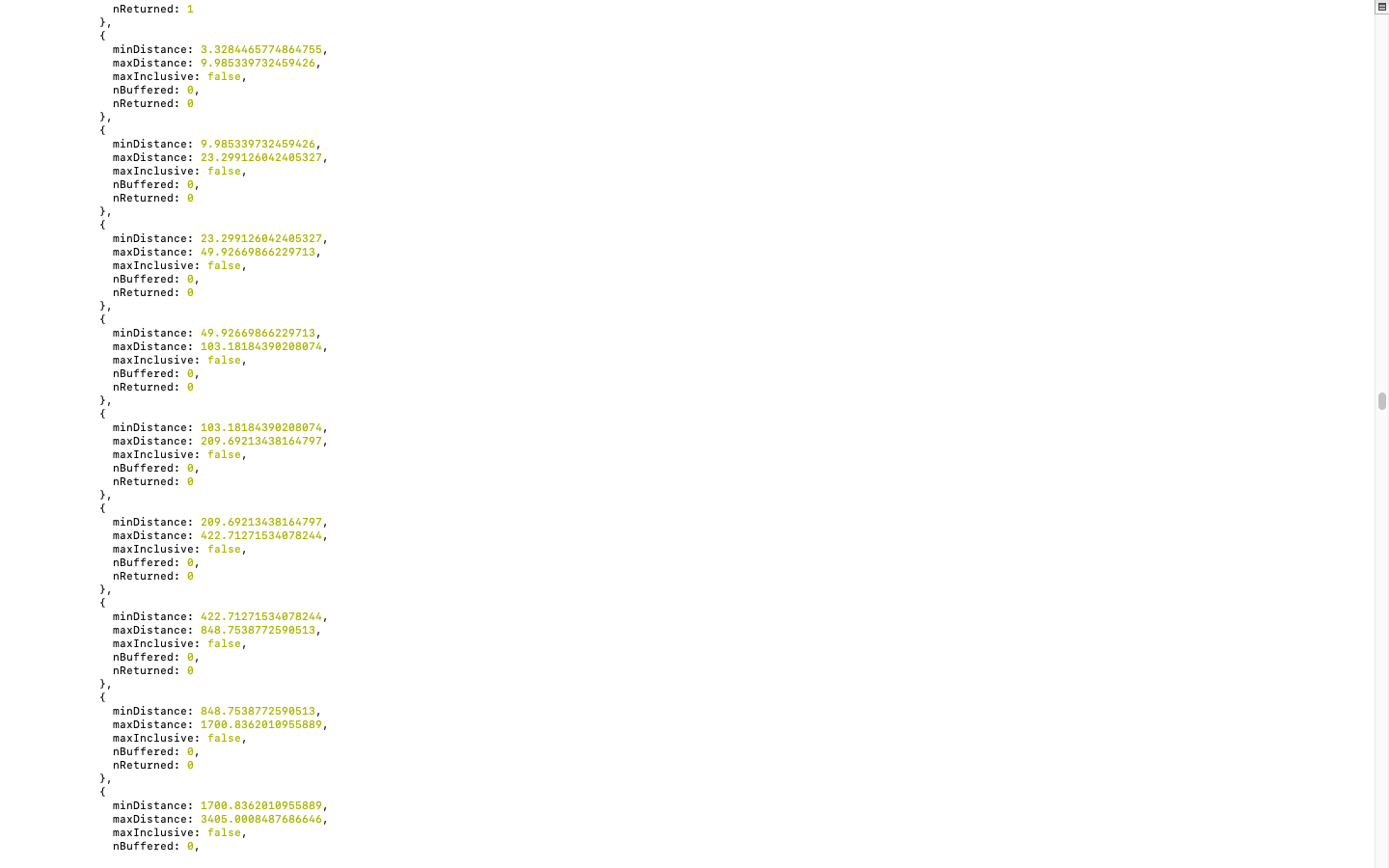
}

}

]).explain()

O/p: The produced o/p is too huge to capture over a screenshot. Hence posting a summary of the output part







Step 3: For analyzing queries involving areas of interest, you can use the explain() method with the find()

db.geodata.find({

location: {

$geoWithin: {

$geometry: {

type: "MultiPolygon",

coordinates: [

[

[

[73.7033602,40.752222],

[74.0051758,40.7112562],

[74.0631544,40.7345715],

[73.7033602,40.752222]

]

],

[

[

[73.9881175,40.7579747],

[73.9992087,40.8700085],

[73.7217817,40.7253564],

[73.9881175,40.7579747]

]

]

]

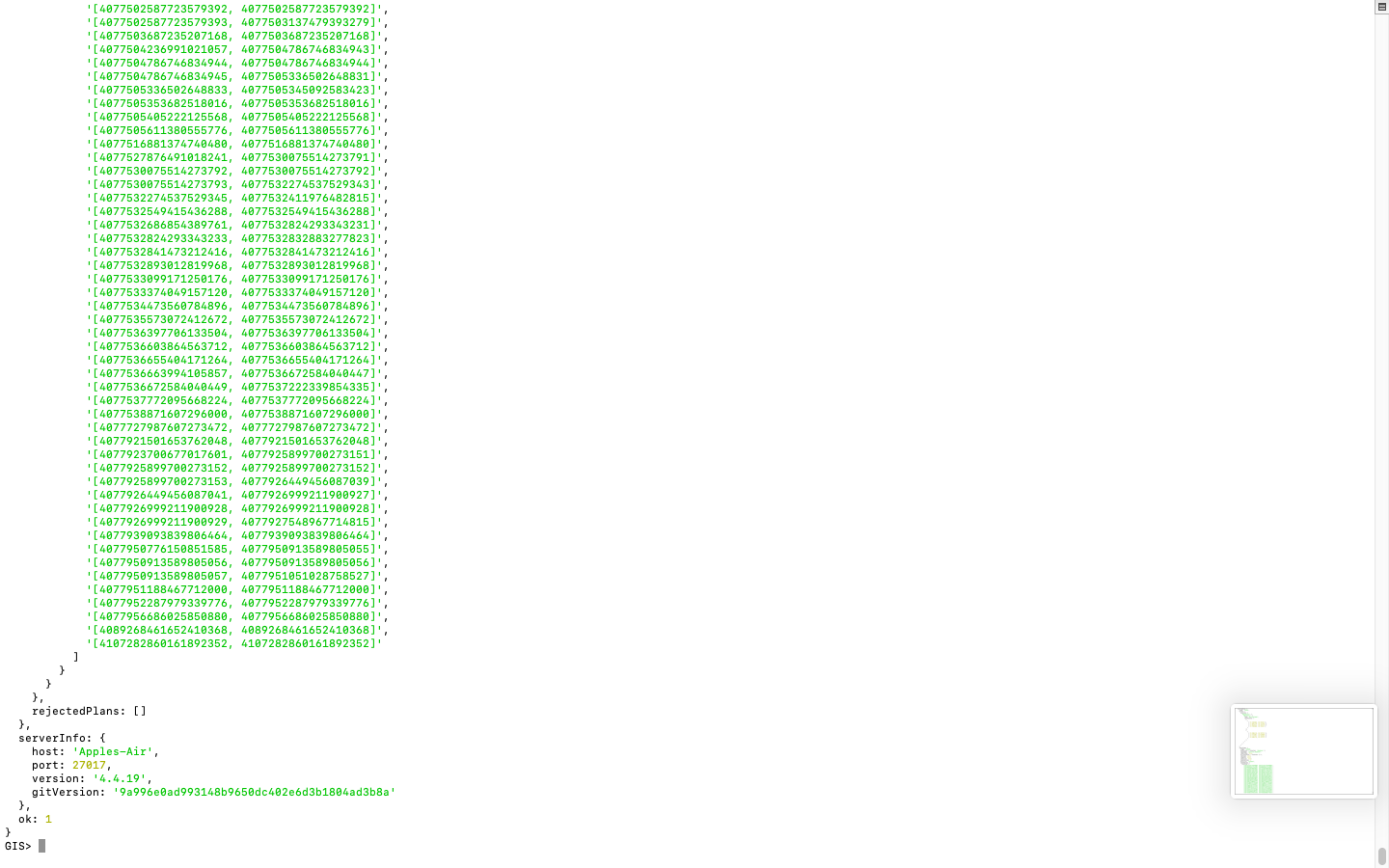
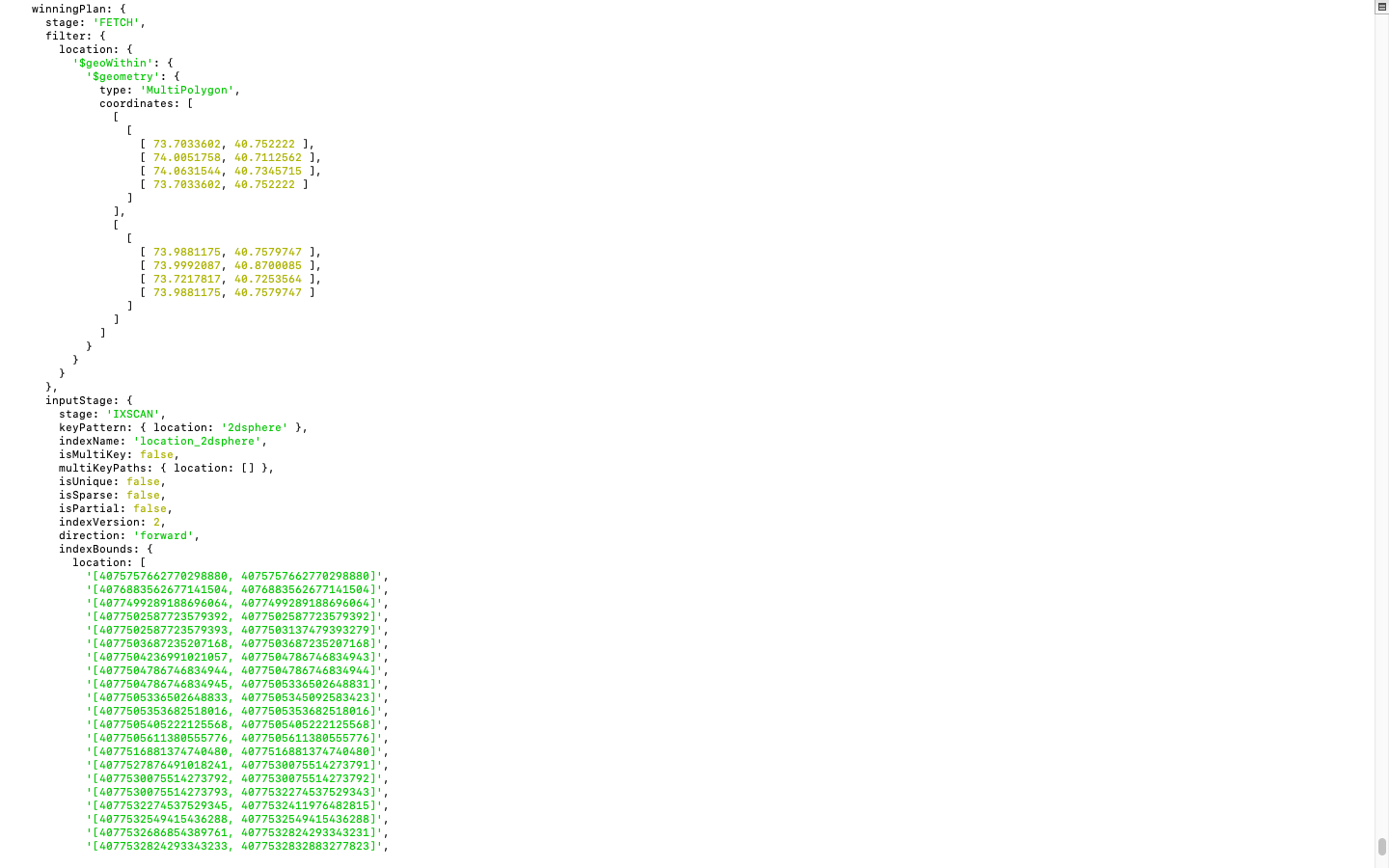
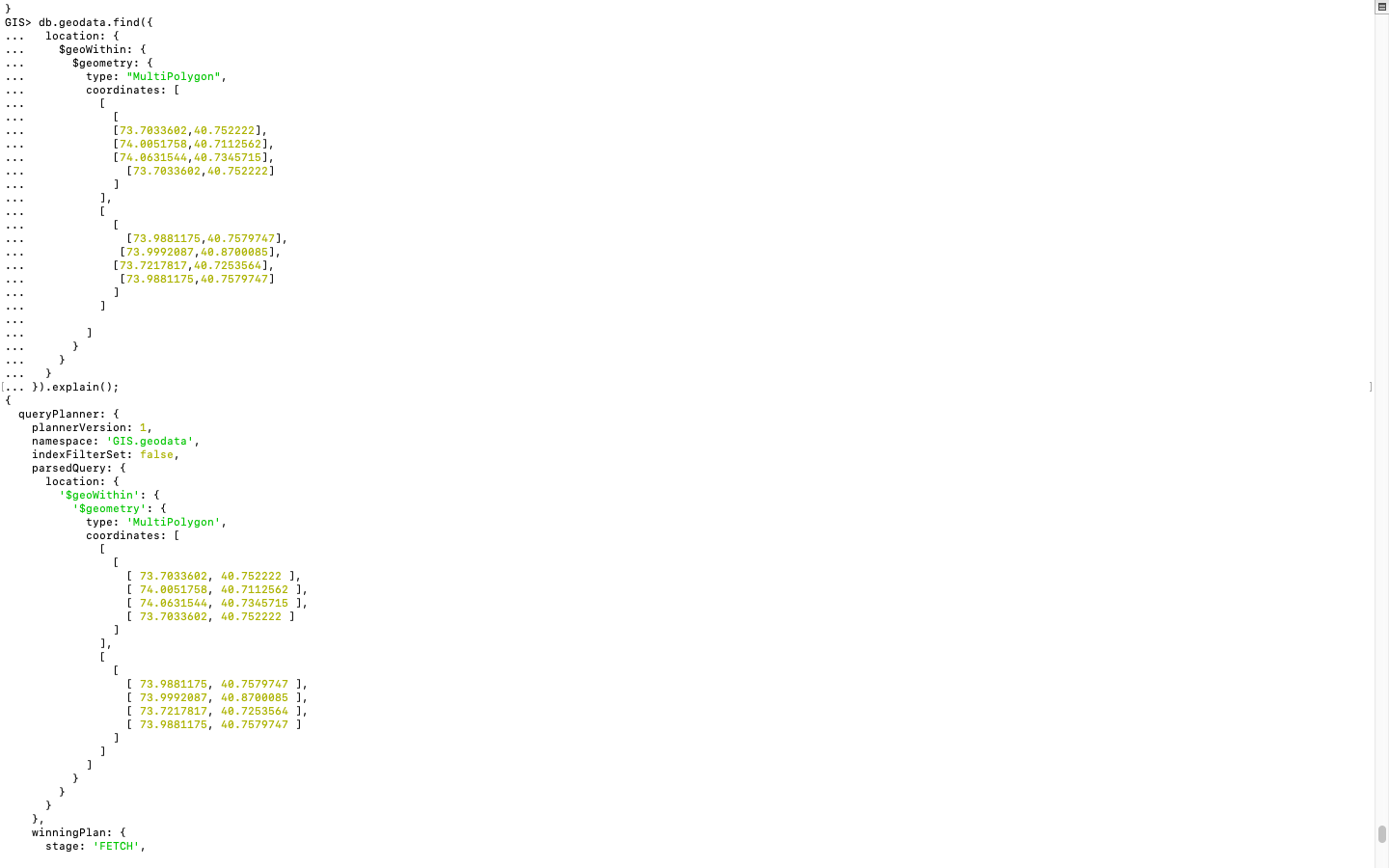
}

}

}

}).explain();

O/p:



5.**Sorting and Limit Executions**

Step1 : Sorting

db.geodata.find({

location: {

$geoWithin: {

$geometry: {

type: "Polygon",

coordinates: [

[

[73.7033602, 40.752222],

[74.0051758, 40.7112562],

[74.0631544, 40.7345715],

[73.9881175, 40.7579747],

[73.9992087, 40.8700085],

[73.7033602, 40.752222] // Closing the loop by repeating the first coordinate

]

]

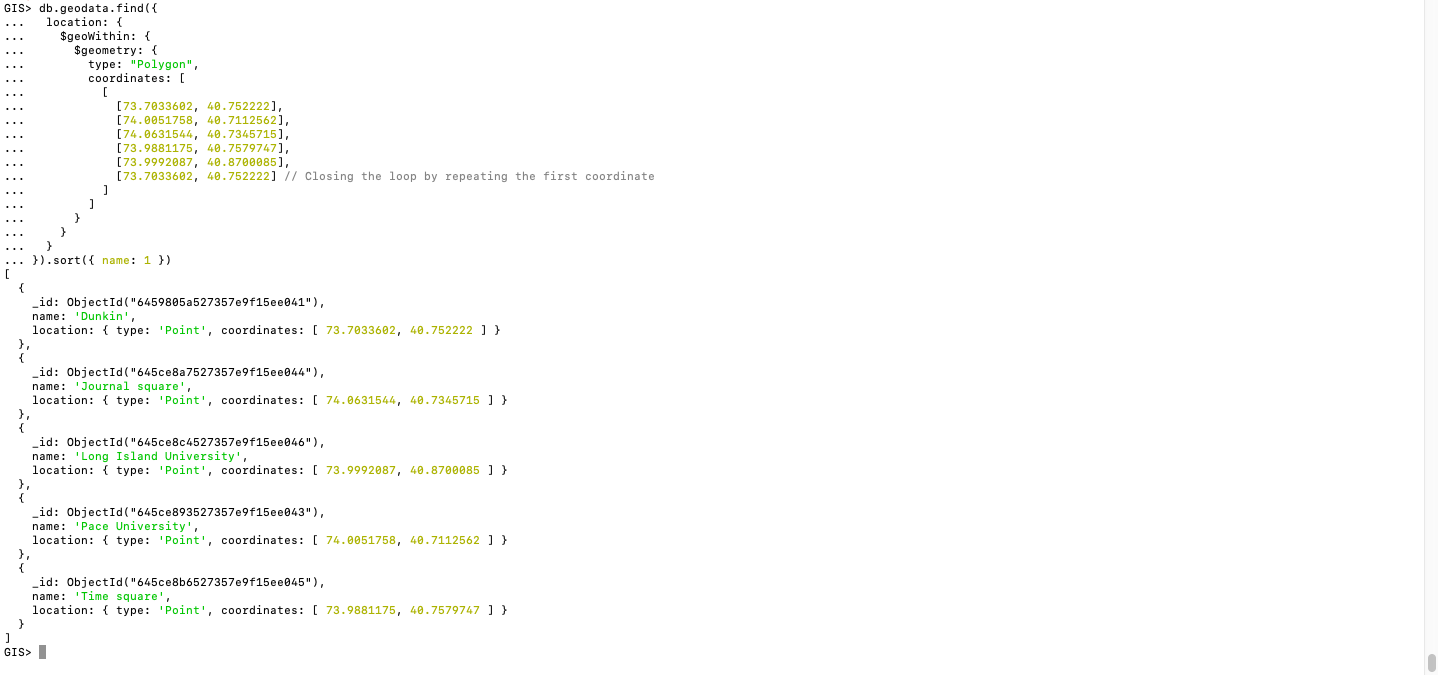
}

}

}

}).sort({ name: 1 })

O/p:



Step 2:

db.geodata.find({

location: {

$geoWithin: {

$geometry: {

type: "Polygon",

coordinates: [

[

[73.7033602, 40.752222],

[74.0051758, 40.7112562],

[74.0631544, 40.7345715],

[73.9881175, 40.7579747],

[73.9992087, 40.8700085],

[73.7033602, 40.752222] // Closing the loop by repeating the first coordinate

]

]

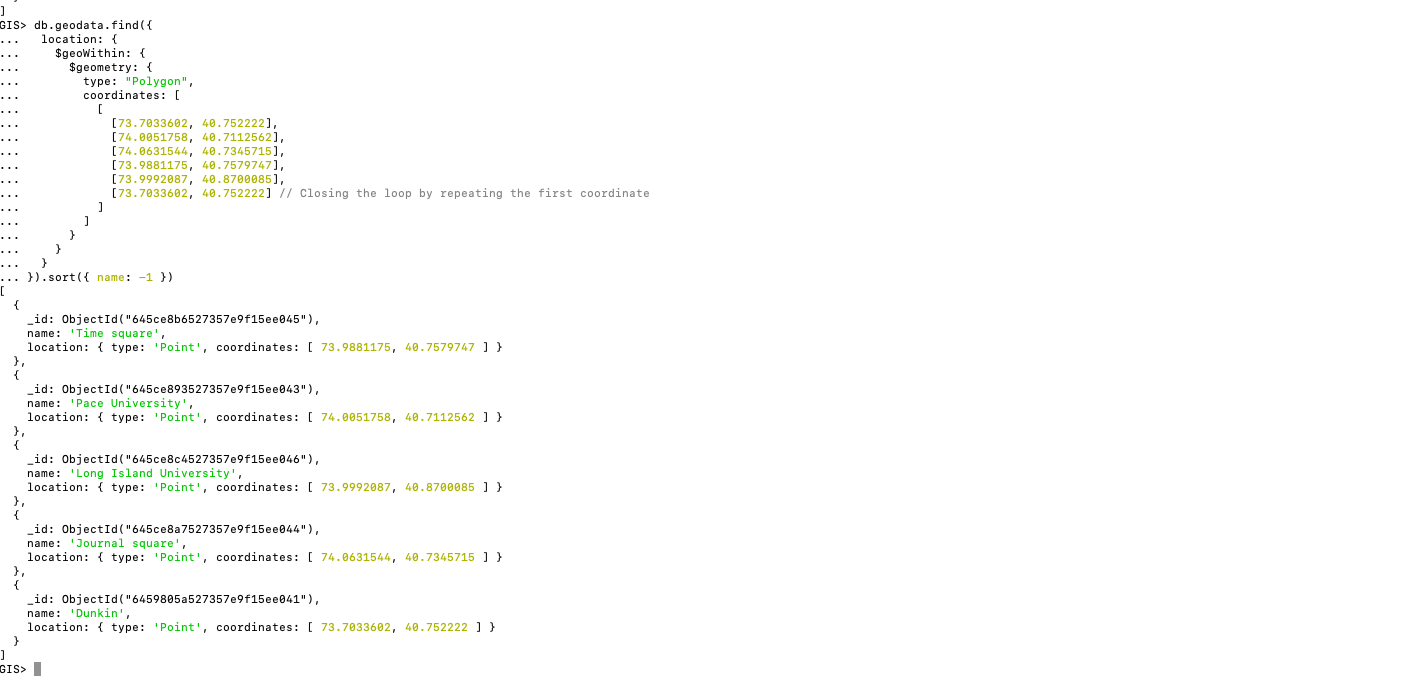
}

}

}

}).sort({ name: -1 })

O/p:



Step 3: Limit the results to 3

db.geodata.find({

location: {

$geoWithin: {

$geometry: {

type: "Polygon",

coordinates: [

[

[73.7033602, 40.752222],

[74.0051758, 40.7112562],

[74.0631544, 40.7345715],

[73.9881175, 40.7579747],

[73.9992087, 40.8700085],

[73.7033602, 40.752222] // Closing the loop by repeating the first coordinate

]

]

}

}

}

}).limit(3)

O/p:



6.**Optimize the queries to speed up execution time**

Create Indexes:

Ensure that you have appropriate indexes on the fields involved in your queries. For geospatial queries, create a geospatial index on the location field. This index will significantly improve the performance of location-based operations. Create the index using the following command:

db.geodata.createIndex({location:'2dsphere'});

O/p:



Query Optimization:

Review your queries and analyze if there are any opportunities for optimization. Look for unnecessary or redundant operations that can be eliminated. Ensure that you are leveraging the available query operators effectively to filter and narrow down the results. For example, if you have a large dataset, consider adding additional filters to limit the search space.

Projection:

Optimize the projection of your query results by specifying only the required fields to be returned. Limiting the number of fields retrieved can improve performance, especially when dealing with large or complex documents. Use the projection parameter to specify the fields to be included or excluded from the result set.

Batch Processing:

If you have a large dataset and your queries are still slow, consider breaking down the queries into smaller batches or chunks. Process the data in smaller subsets to reduce the overall load on the database and improve performance.

Hardware and Configuration:

Ensure that your MongoDB server is properly configured and adequately provisioned in terms of hardware resources such as CPU, memory, and storage. Tune the MongoDB configuration parameters based on your workload and system requirements to optimize performance.

Consider Sharding:

If you anticipate a significant increase in data volume and query load, consider sharding your MongoDB cluster. Sharding distributes the data across multiple servers, allowing for horizontal scalability and improved performance for large-scale deployments.

7**.N-Optimization of queries**

Aggregation Pipeline Optimization:

If you are using the aggregation pipeline for complex queries, analyze the stages and operations within the pipeline for potential optimization. Look for opportunities to leverage available operators, such as $match, $group, $project, and $sort, to streamline the pipeline and reduce unnecessary operations.

Query Rewriting:

Consider rewriting your queries to leverage more efficient operations or to simplify complex queries. Sometimes, a different query structure or approach can result in significant performance improvements.

Use $exists to Optimize Index Use:

If you are querying for documents based on the presence or absence of a field, consider using the $exists operator. $exists can help optimize index use, reducing the number of documents that need to be scanned to satisfy a query.

Avoid Negation in Queries:

Avoid using the $not operator or negating conditions in your queries. Negation requires a full table scan, which can be a performance bottleneck for large datasets. Instead, consider restructuring your queries to avoid negation.

Query Profiling:

Enable query profiling in MongoDB to identify and analyze slow queries. Use the db.setProfilingLevel() command to enable profiling at an appropriate level, review the profiling data, and optimize the identified queries.

**Additionally we can use the .explain() method to understand the query execution plan, index usage, and performance statistics.**