MONGO DB

EXPERIMENT-03

3.a. Execute query selectors (comparison selectors, logical selectors) and list out the results on any collection.

b. Execute query selectors (Geospatial selectors, Bitwise selectors) and list out the results on any collection.

MongoDB offers various query selectors to filter documents based on specific criteria. Here, we'll explore comparison and logical selectors

A. Comparison and Logical Selectors

1.Comparison Selectors:

These operators compare field values with a specified value or range.

These selectors compare field values with specific operators:

- $eq: Matches documents where a field equals a specific value (e.g., `age: { $eq: 30 }`).
- $gt: Matches documents where a field is greater than a value (e.g., `price: { $gt: 10 }`).
- $lt: Matches documents where a field is less than a value (e.g., `quantity: { $lt: 5 }`).
- $gte: Matches documents where a field is greater than or equal to a value (e.g., `score: { $gte: 80 }`).
- $lte: Matches documents where a field is less than or equal to a value (e.g., `arrival_date: { $lte: ISODate("2024-06-10") }`).
- $in: Matches documents where a field's value is present in an array of specified values (e.g., `category: { $in: ["Electronics", "Clothing"] }`).
- $nin: Matches documents where a field's value is not present in an array of specified values (e.g., `status: { $nin: ["Completed", "Cancelled"] }`).

Example 01:

Retrieve the data who all have gpa equal to 3.5

```
db> db.student.find({ gpa:{$eq: 3.5}}, { name: 1,gpa:1,_id:0}).limit(6);
[
  { name: 'Fatima Brown', gpa: 3.5 },
  { name: 'Lily Robinson', gpa: 3.5 }
]
db>
```

Example 02:

Retrieve the data who all have gpa not equal to 3.5

```
db> db.student.find({ gpa:{$ne: 3.5}}, { name: 1,gpa:1,_id:0}).limit(6);
[
  { name: 'Alice Smith', gpa: 3.4 },
  { name: 'Bob Johnson', gpa: 3.8 },
  { name: 'Charlie Lee', gpa: 3.2 },
  { name: 'Emily Jones', gpa: 3.6 },
  { name: 'David Williams', gpa: 3 },
  { name: 'Gabriel Miller', gpa: 3.9 }
]
db>
```

Example 03:

Retrieve the data who all have gpa greater than or equal to 3.5

```
db> db.student.find({ gpa:{$gte: 3.5}}, { name: 1,gpa:1,_id:0}).limit(6);
[
  { name: 'Bob Johnson', gpa: 3.8 },
  { name: 'Emily Jones', gpa: 3.6 },
  { name: 'Fatima Brown', gpa: 3.5 },
  { name: 'Gabriel Miller', gpa: 3.9 },
  { name: 'Isaac Clark', gpa: 3.7 },
  { name: 'Kevin Lewis', gpa: 4 }
]
db>
```

Example 04:

Retrieve the data who all have gpa greater than 3.5

```
db> db.student.find({ gpa:{$gt: 3.5}}, { name: 1,gpa:1,_id:0}).limit(6);
[
  { name: 'Bob Johnson', gpa: 3.8 },
  { name: 'Emily Jones', gpa: 3.6 },
  { name: 'Gabriel Miller', gpa: 3.9 },
  { name: 'Isaac Clark', gpa: 3.7 },
  { name: 'Kevin Lewis', gpa: 4 }
]
```

Example 05:

Retrieve the data who all have gpa less than or equal to 3.5.

```
db> db.student.find({ gpa:{$lte: 3.5}}, { name: 1,gpa:1,_id:0}).limit(6);
[
  { name: 'Alice Smith', gpa: 3.4 },
  { name: 'Charlie Lee', gpa: 3.2 },
  { name: 'David Williams', gpa: 3 },
  { name: 'Fatima Brown', gpa: 3.5 },
  { name: 'Hannah Garcia', gpa: 3.3 },
  { name: 'Jessica Moore', gpa: 3.1 }
]
db>
```

Example 06:

Retrieve the data who all have gpa less than 3.5.

```
db> db.student.find({ gpa:{$lt: 3.5}}, { name: 1,gpa:1,_id:0}).limit(6);
[
  { name: 'Alice Smith', gpa: 3.4 },
  { name: 'Charlie Lee', gpa: 3.2 },
  { name: 'David Williams', gpa: 3 },
  { name: 'Hannah Garcia', gpa: 3.3 },
  { name: 'Jessica Moore', gpa: 3.1 }
]
db>
```

2. Logical Selectors:

These operators combine multiple comparison conditions:

- $and: Matches documents where all specified conditions are true (e.g., `{ age: { $gt: 21 }, city: "New York" }`).
- $or: Matches documents where at least one specified condition is true (e.g., `{ $or: [{ price: { $lt: 50 } }, { category: "Books" }] }`).
- $not: Matches documents where the specified condition is false (e.g., `{ active: { $not: true } }`).

Example 01:

Retrieve students from city 2 with blood group “B+”.

```
db.students.find({
  $and: [
    { home_city: "City 2" },
    { blood_group: "B+" }
  ]
});
```

Example 02:

Retrieve students from hotel resident OR have a gpa less than 3.

```
db.students.find({
  $or: [
    { is_hotel_resident: true },
    { gpa: { $lt: 3.0 } }
  ]
});
```

B. Query Selectors (Geospatial & Bitwise)

1.Geospatial Selectors:

These operators are used for spatial queries involving geospatial data stored using GeoJSON format. They require a geospatial index on the relevant field. Common examples include:

$geoWithin:

Matches documents where a geospatial field intersects a specified GeoJSON geometry (e.g., a bounding box).

$geoIntersects:

Matches documents where a geospatial field intersects another geospatial field.

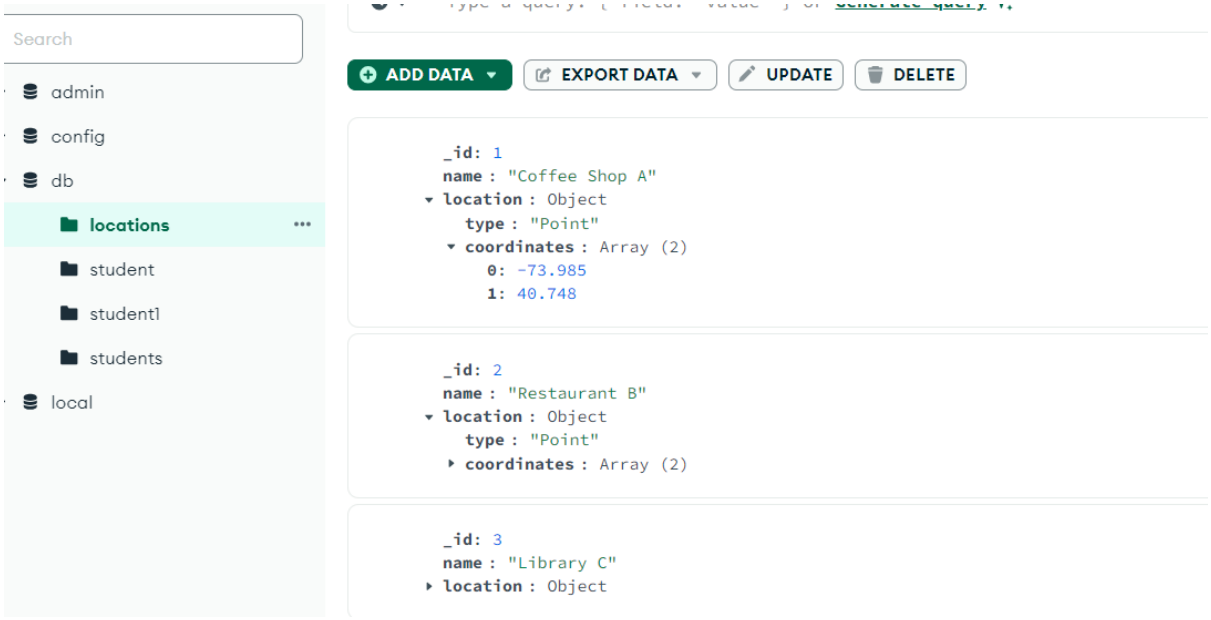
$near:

Matches documents within a specified spherical distance from a given point.

Geospatial Query Operators

Name	Description
`$geoIntersects`	Selects geometries that intersect with a GeoJSON geometry. The `2dsphere` index supports `$geoIntersects`.
`$geoWithin`	Selects geometries within a bounding GeoJSON geometry. The `2dsphere` and `2d` indexes support `$geoWithin`.
`$near`	Returns geospatial objects in proximity to a point. Requires a geospatial index. The `2dsphere` and `2d` indexes support `$near`.
`$nearSphere`	Returns geospatial objects in proximity to a point on a sphere. Requires a geospatial index. The `2dsphere` and `2d` indexes support `$nearSphere`.

Adding a new json file to database with collection name -locations.



Example 01:

1 kilometer in radius

```
db> db.locations.find({
... location:{
... $geoWithin:{
... $centerSphere:[[-74.005,40.712],0.00621376]
... }
... }
... });
[
  {
    _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 ] }
  }
]
db>
```

2.Bitwise Selectors:

These operators perform bitwise operations on integer values in documents. However, their use is generally discouraged due to potential performance limitations and the availability of alternative approaches. Common examples include:

$and: Performs a bitwise AND operation on two integer values.

$or: Performs a bitwise OR operation on two integer values.

Adding a new CSV file to database with collection name - students_permission.

```
_id: ObjectId('66686179cbb422f46f349188')
name : "Alice"
age : 22
permissions : 0
```

```
_id: ObjectId('66686179cbb422f46f349189')
name : "Bob"
age : 25
permissions : 1
```

```
_id: ObjectId('66686179cbb422f46f34918a')
name : "Charlie"
age : 20
permissions : 2
```

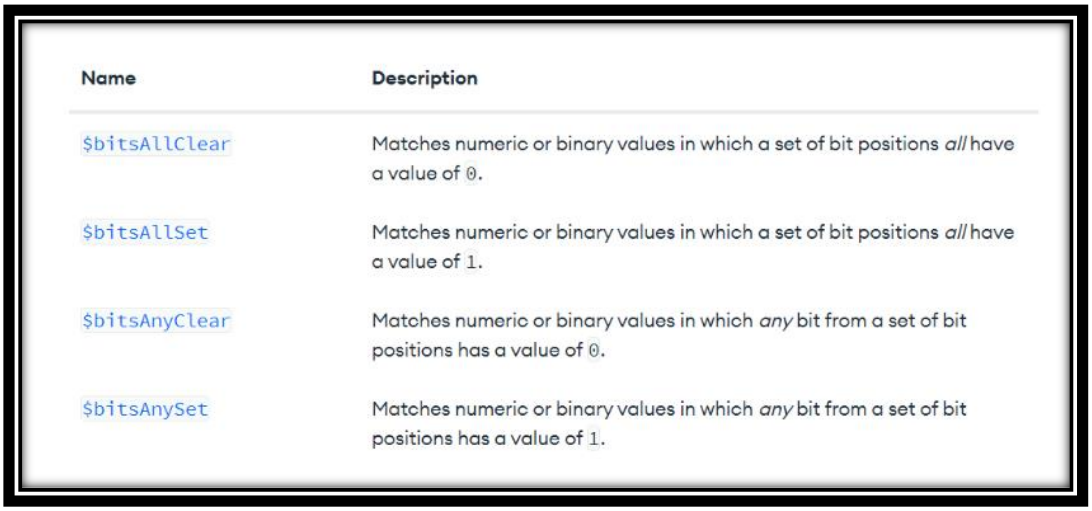
Limitations and Alternatives:

Geospatial Selectors:

- ✓ Require a geospatial index on the field you're querying.
- ✓ May have performance implications for complex queries.

Bitwise Selectors:

- ✓ Not commonly used due to potential performance issues.
- ✓ Consider using alternative approaches like conditional logic or custom functions for complex bitwise operations.



Name	Description
`$bitsAllClear`	Matches numeric or binary values in which a set of bit positions *all* have a value of `0`.
`$bitsAllSet`	Matches numeric or binary values in which a set of bit positions *all* have a value of `1`.
`$bitsAnyClear`	Matches numeric or binary values in which *any* bit from a set of bit positions has a value of `0`.
`$bitsAnySet`	Matches numeric or binary values in which *any* bit from a set of bit positions has a value of `1`.

Example 01: Bit Positions for permissions

```
db> const LOBBY_PERMISSION=1;

db> const CAMPUS_PERMISSION=2;

db> db.students_permission.find({
... permissions:{$bitsAllSet:[LOBBY_PERMISSION,CAMPUS_PERMISSION]}
... });
[
  {
    _id: ObjectId('66686179cbb422f46f34918e'),
    name: 'George',
    age: 21,
    permissions: 6
  },
  {
    _id: ObjectId('66686179cbb422f46f34918f'),
    name: 'Henry',
    age: 27,
    permissions: 7
  },
  {
    _id: ObjectId('66686179cbb422f46f349190'),
    name: 'Isla',
    age: 18,
    permissions: 6
  }
]
db>
```

While these selectors can be useful in specific scenarios, it's important to understand their limitations and consider alternative approaches when appropriate.

Datatype

MongoDB itself doesn't have specific data types for geometric shapes like Point, LineString, or Polygon. However, it leverages the GeoJSON format to store geospatial data.

GeoJSON is a JSON extension that allows representing various geographic features including points, linestrings, and polygons. Here's a brief explanation of each within GeoJSON context:

- **Point:** Represents a single geographic location using latitude and longitude coordinates. In GeoJSON, it's defined as an object with "type" set to "Point" and a "coordinates" array containing [longitude, latitude].
- **LineString:** Represents a sequence of connected geographic positions. It's defined as a GeoJSON object with "type" set to "LineString" and a "coordinates" array containing multiple longitude, latitude pairs.
- **Polygon:** Represents a closed area defined by a sequence of connected geographic positions. A valid polygon must start and end at the same point. In GeoJSON, it's defined as an object with "type" set to "Polygon" and a "coordinates" array. This array contains one outer ring (defining the polygon boundary) and can optionally include inner rings (representing holes within the polygon). Each ring is itself an array of longitude, latitude pairs.

By storing GeoJSON documents within MongoDB documents, you can effectively store and query geospatial data.