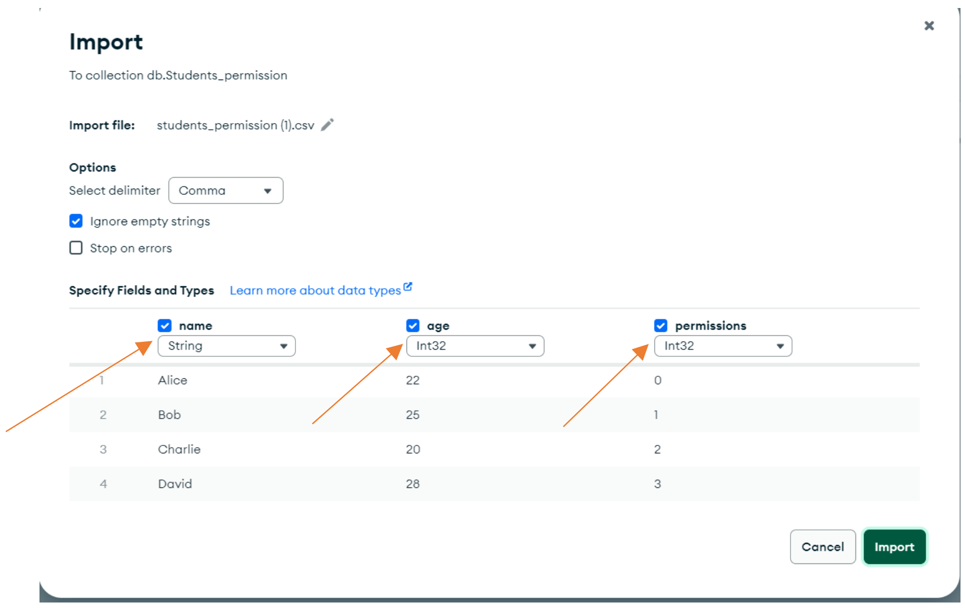
MongoDB

BITWISE,QUERY & GEOPATIAL

BITWISE:

MongoDB provides functionalities for manipulating bits within integer fields (32-bit or 64-bit) using bitwise operators. These operators are especially useful for scenarios where you need to perform flag-based operations or manipulate specific bit positions.


Import

To collection db.Students_permission

Import file: students_permission (1).csv

Options

Select delimiter: Comma

☑ Ignore empty strings

☐ Stop on errors

Specify Fields and Types — Learn more about data types

	☑ name String	☑ age Int32	☑ permissions Int32
1	Alice	22	0
2	Bob	25	1
3	Charlie	20	2
4	David	28	3

Here's a detailed explanation of bitwise operators in MongoDB:

1. Bitwise Update Operator ($bit):

This operator allows you to perform bitwise AND, OR, and XOR operations on existing integer fields within a document. You can leverage it within update methods like **update()** and **findAndModify().**

Syntax:

{ $bit: { <field>: { <and|or|xor>: <int> } } }

<field>: The integer field you want to perform the bitwise operation on.

<and|or|xor>: Specifies the type of bitwise operation (AND, OR, or XOR).

<int>: The integer value used for the bitwise operation.

Examples:

Bitwise AND: Set specific bits in a field to 0 (clear them).

```
db.users.updateOne(
   { _id: 1 },
   { $bit: { permissions: { and: NumberInt(14) } } } // 14 in binary is 00001110
)
```

Bitwise OR: Set specific bits in a field to 1 (set them).

```
db.users.updateOne(
   { _id: 1 },
   { $bit: { permissions: { or: NumberInt(3) } } } // 3 in binary is 00000011
)
```

Bitwise XOR: Flip the bits in a field.

```
db.users.updateOne(
   { _id: 1 },
   { $bit: { permissions: { xor: NumberInt(7) } } } // 7 in binary is 00000111
)
```

2. Bitwise Query Operators:

MongoDB offers several query operators to filter documents based on specific bit patterns within an integer field. These operators are particularly useful for finding documents with specific flag combinations.

Operators:

$bitsAllSet: Matches documents where all specified bits are set to 1 (active).

```
db.users.find({ permissions: { $bitsAllSet: NumberInt(12) } }) // Checks if bits 2 and 3 are set
```

$bitsAllClear: Matches documents where all specified bits are set to 0 (inactive).

```
db.users.find({ permissions: { $bitsAllClear: NumberInt(6) } }) // Checks if bits 1 and 2 are clear
```

$bitsAnySet: Matches documents where any of the specified bits are set to 1.

```
db.users.find({ permissions: { $bitsAnySet: NumberInt(5) } }) // Checks if bit 0 or 2 is set
```

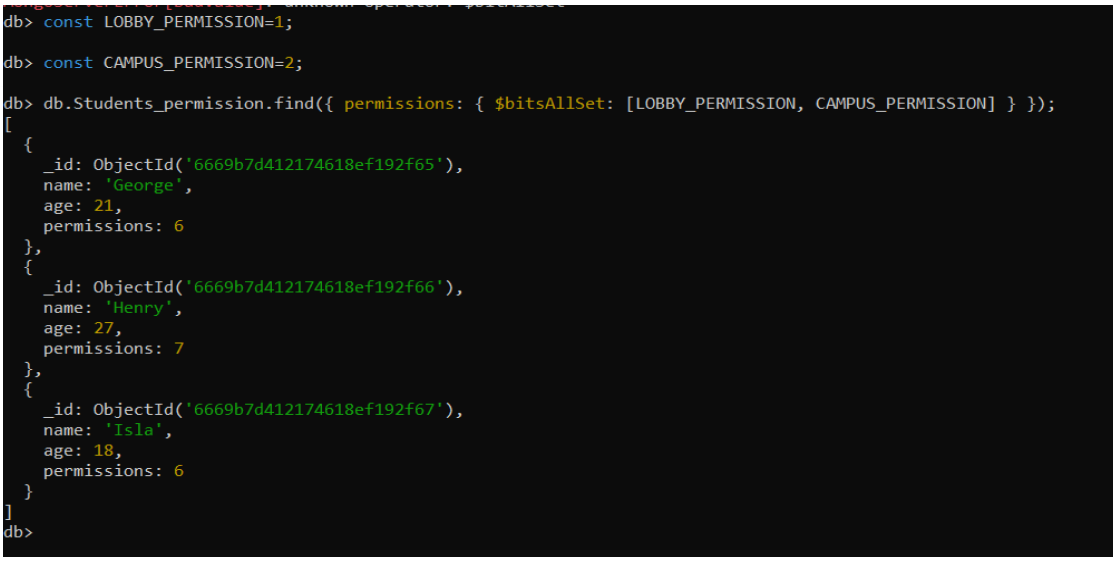
$bitsAnyClear: Matches documents where any of the specified bits are set to 0.

```
db.users.find({ permissions: { $bitsAnyClear: NumberInt(14) } }) // Checks if any of bits 1, 2 or 3 are clear
```

QUERY:

Constructing Queries for Lobby and Campus Permissions in MongoDB using const

Sure, here's a detailed explanation on how to construct queries for lobby and campus permissions in MongoDB using constants:



```
db> const LOBBY_PERMISSION=1;

db> const CAMPUS_PERMISSION=2;

db> db.Students_permission.find({ permissions: { $bitsAllSet: [LOBBY_PERMISSION, CAMPUS_PERMISSION] } });
[
  {
    _id: ObjectId('6669b7d412174618ef192f65'),
    name: 'George',
    age: 21,
    permissions: 6
  },
  {
    _id: ObjectId('6669b7d412174618ef192f66'),
    name: 'Henry',
    age: 27,
    permissions: 7
  },
  {
    _id: ObjectId('6669b7d412174618ef192f67'),
    name: 'Isla',
    age: 18,
    permissions: 6
  }
]
db>
```

- Two lines define constants, LOBBY_PERMISSION_BIT and CAMPUS_PERMISSION_BIT, which are seemingly not used in the query shown here. These constants likely represent bit positions within a document field (possibly for permissions) based on their names.
- The main query part is:

  db.Students.find((permission:{(Shitsalisel:LOBBY_PERMISSION,CAMPUS_PERMISSION) }))

- This query seems to be targeting a collection named "Students" and applying a filter based on a field named "permission". However, there are syntax errors and a logical issue:

- **Missing Field Value:** The query uses a colon (:) after `permission` but lacks a value to compare with. It's unclear what value was intended to be matched against the "permission" field.
- **Incorrect Subdocument Syntax:** The subdocument syntax within the query appears malformed. The intended structure for checking multiple bits with bitwise operators is likely incorrect.

GEOPATIAL:

In MongoDB, geospatial queries allow you to search for data based on its location. This is particularly useful for applications that deal with geospatial data, like finding nearby restaurants, managing delivery zones, or analyzing user activity across locations. In MongoDB, geospatial queries allow you to search for data based on its location. This is particularly useful for applications that deal with geospatial data, like finding nearby restaurants, managing delivery zones, or analyzing user activity across locations.

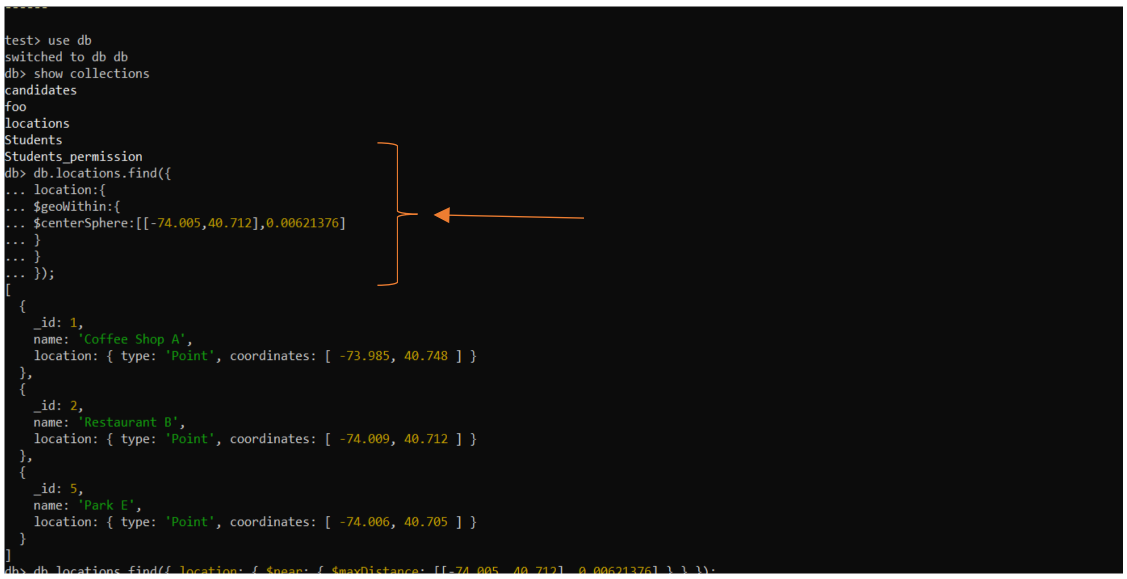
Example GeoJSON Point:

{ "type": "Point", "coordinates": [-73.985, 40.748] }

This represents a point at longitude -73.985 and latitude 40.748.

Geospatial Operator	Description	Syntax
$near	Finds geospatial objects near a point. Requires a geospatial index.	{ $near: { geometry: <point_geometry>, maxDistance: <distance> (optional) } }
$center	(For $geoWithin with planar geometry) Specifies a circle around a center point	{ $geoWithin: { $center: [<longitude>, <latitude>], radius: <distance> } }
$maxDistance	Limits results of $near and $nearSphere queries to a maximum distance from the point.	{ $near: { geometry: <point_geometry>, maxDistance: <distance> } }
$minDistance	Limits results of $near and $nearSphere queries to a minimum distance from the point.	{ $near: { geometry: <point_geometry>, minDistance: <distance> } }

Example:



```
test> use db
switched to db db
db> show collections
candidates
foo
locations
Students
Students_permission
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> db.locations.find({ location: { $near: { $maxDistance: [[-74.005, 40.712], 0.00621376] } } });
```

The provided code snippet represents a MongoDB query searching for documents within a specific geospatial area using the $geoWithin operator and the 2dsphere index. Let's break it down:

Database and Collection: The query targets the locations collection within the current database (assuming you've already connected to the database).

Query Filter: The find method applies a filter to identify matching documents. The filter focuses on the location field within each document.

Geospatial Search:

$geoWithin: This operator specifies that the location field should be searched based on its geospatial relationship with another geometry.

$centerSphere: This indicates that we're using the 2dsphere geospatial index (suitable for spherical Earth-like data) and searching for locations within a circle centered at a specific point.

[-74.005, 40.712]: This defines the center point of the circle. In this case, it's longitude -74.005 and latitude 40.712.

0.00621376: This value represents the radius of the circle in radians. Here, it's approximately 0.35 miles (assuming the coordinates are in degrees).

DATATYPE:

In MongoDB, geospatial data can be represented using GeoJSON, a standardized format for encoding geographic features. Here's an explanation of the three data types you mentioned for geospatial data in MongoDB:

1. Point:

- Represents a single geographic location with a longitude and latitude.

  GeoJSON Example:

  ```
  {
   "type": "Point",

   "coordinates": [longitude, latitude]
  }
  ```

  Example Usage: Storing the location of a store, a sensor, or a user's current location.

2. LineString:

- Represents a sequence of connected coordinate pairs, defining a linear path.

  GeoJSON Example:

```
{
  "type": "LineString",
  "coordinates": [
    [longitude1, latitude1],
    [longitude2, latitude2],
    ... // Additional coordinate pairs for the path
  ]
}
```

- **Example Usage:** Storing the route of a delivery truck, the outline of a river, or a hiking trail.

3. Polygon:

- Represents a closed loop defined by an array of coordinate pairs, outlining an area on the map.
- The first and last coordinate pairs should be the same to close the loop.

- **GeoJSON Example:**

```
{
  "type": "Polygon",
  "coordinates": [
    [ // Array of coordinate pairs for each vertex of the polygon
      longitude1, latitude1
    ],
    [ ... ],
    [ longitudeN, latitudeN ] // Closing vertex (same as the first one)
  ]
}
```

- **Example Usage:** Storing the boundary of a city, a national park, or a building footprint.