Where, And, Or and Crud:

Where:

In MongoDB, the $where operator allows you to specify a JavaScript expression or function to perform more complex queries that cannot be expressed using the standard MongoDB query language syntax. This operator evaluates JavaScript code on the MongoDB server for each document in the collection.

Here's a basic example of how you might use the $where operator:

```
db.collection.find({
    $where: function() {
        return this.age > 30 && this.age < 50;
    }
})
```

In this example, db.collection is the collection you're querying, and the $where operator is used to evaluate a JavaScript function for each document. The function returns true for documents where the age field is greater than 30 and less than 50.

It's important to note that using $where can have performance implications, as it may require scanning the entire collection and executing JavaScript code for each document. It's generally recommended to use the standard MongoDB query operators whenever possible for better performance.

And:

In MongoDB, the $and operator allows you to combine multiple expressions within the same query, where all conditions must be met for a document to match. Here's a basic example:

```
db.collection.find({
    field1: value1,
    field2: value2
})
```

In this example, db.collection is the collection you're querying. The $and operator is used to specify that both conditions must be met. Each condition is a separate document within the array passed to $and.

You can include as many conditions as needed within the $and array. MongoDB will evaluate each condition and return documents where all conditions are true.

It's worth noting that you can omit the $and operator and MongoDB will interpret multiple conditions at the same level of nesting as an implicit "AND" operation. So the above example could also be written without $and:

Both forms are equivalent and will return documents where both field1 equals value1 and field2 equals value2.

Or:

In MongoDB, the $or operator allows you to specify multiple conditions, where at least one condition must be met for a document to match. Here's an example of how you can use the $or operator to find all students who are hotel residents:

```
db.students.find({
    $or: [
        { residence: "Hotel" },
        { accommodation: "Hotel" }
    ]
})
```

In this example, db.students is the collection you're querying. The $or operator is used to specify that a student must have either "Hotel" as their residence or "Hotel" as their accommodation.

This query will return all documents where either condition is true, meaning all students who are either residents of a hotel or currently accommodated in a hotel. Adjust the field names according to your schema if they are different.

CRUD:

In MongoDB, CRUD (Create, Read, Update, Delete) operations are fundamental for interacting with data. Here's how you perform each CRUD operation in MongoDB:

Create (Insert)

To create (or insert) new documents into a collection, you can use the insertOne() method to insert a single document or insertMany() to insert multiple documents at once.

Example of inserting a single document:

```
db.students.insertOne({
    "name": "Alice",
    "age": 20,
    "grade": "A"
});
```

Read (Query)

To query and retrieve student data from the "students" collection:

```
// Find all students
db.students.find();

// Find students with a specific name
db.students.find({ "name": "Alice" });

// Find students with age less than 25
db.students.find({ "age": { $lt: 25 } });
```

Update

To update existing student data in the "students" collection:

```
db.students.updateOne(
    { "name": "Alice" },
    { $set: { "age": 21 } }
);
```

Delete:

To delete student data from the "students" collection:

```
// Delete a student
db.students.deleteOne({ "name": "Alice" });
```

These examples demonstrate how you can perform CRUD operations on a MongoDB collection containing student data represented as JSON documents.

Update Many:

Suppose we want to update the grade of all students with an age greater than 20 to "B":

```
db.students.updateMany(
    { "age": { $gt: 20 } },
    { $set: { "grade": "B" } }
);
```

This command updates the "grade" field of all documents in the "students" collection where the "age" is greater than 20 to "B".

Delete Many:

Suppose we want to delete all students with a grade of "C":

```
db.students.deleteMany(
    { "grade": "C" }
);
```

This command deletes all documents from the "students" collection where the "grade" is "C".

These examples demonstrate how to use updateMany() and deleteMany() operations in MongoDB to perform bulk updates and deletions on a collection of student documents.

Projection,Limit,and Selectors:

Projection:

projections are used to control which fields are returned in query results. Projections allow you to include or exclude specific fields, making queries more efficient by returning only the necessary data.

1.Including Specific Fields: To include only specific fields in the output, use the find method with a projection document:

```
db.students.find(
    { "grade": "A" },  // Query condition
    { "name": 1, "age": 1, "_id": 0 }  // Projection
);
```

2.Excluding Specific Fields: To exclude specific fields, set the fields to 0 in the projection document:

```
db.students.find(
    { "grade": "A" },  // Query condition
    { "address": 0 }  // Projection
);
```

This command will return documents where the grade is "A", but the address field will be excluded from the output. The _id field is included by default unless explicitly excluded.

Using Projections with Nested Documents:

Projections can also be used with nested documents by specifying the path to the nested fields.

```
db.students.find(
    { "grade": "A" },
    { "name": 1, "contact.phone": 1, "_id": 0 }
);
```

Using projections in MongoDB has several benefits that can significantly enhance the efficiency and performance of your database queries. Here are the key benefits:

1. Improved Performance

By specifying only the fields you need in your query results, projections can reduce the amount of data that MongoDB needs to retrieve and transmit over the network. This can lead to faster query execution times and reduced network latency.

2. Reduced Memory Usage

When fewer fields are returned, the memory usage on both the MongoDB server and the client application is reduced. This is particularly beneficial when working with documents that have a large number of fields or when querying large collections.

3. Lower Network Bandwidth

Projections help minimize the amount of data sent over the network by limiting the fields returned in the query results. This is especially important in distributed systems or applications with limited bandwidth.

4. Enhanced Application Performance

By fetching only the necessary data, your application can process and render responses more quickly. This is especially useful for web and mobile applications where performance and responsiveness are critical.

5. Increased Security and Privacy

Projections can help enforce data security and privacy by restricting sensitive information from being included in query results. This way, only the necessary fields are exposed to the client application.

6. Simplified Data Handling

Fetching only the relevant fields can simplify the data handling logic in your application. This reduces the complexity of processing and

manipulating large documents, making your codebase cleaner and more maintainable.

7. Optimized Index Usage

When combined with appropriate indexing strategies, projections can help MongoDB optimize query execution. By limiting the fields returned, MongoDB can more effectively use indexes to locate and return the required data.

Limit:

the limit() method is used to specify the maximum number of documents to return in a query result. This is particularly useful for managing the size of the result set and is often used in conjunction with pagination

Basic Usage

To use the limit operator in a query, you simply chain it to the find method like this:

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

Examples

To retrieve the first 6 documents from the students collection:

```
db.students.find(
    {},
    { name: 1, age: 1, _id: 0 }
).sort({ age: -1 }).limit(6)
```

```
// Retrieve the first 6 documents, sorted by age in ascending order
// Include only the name and age fields in the result, excluding _id
db.students.find(
    {},                         // No query condition
    { "name": 1, "age": 1, "_id": 0 }  // Projection to include/exclude specific fields
).sort({ "age": 1 }).limit(6)
```

Selectors:

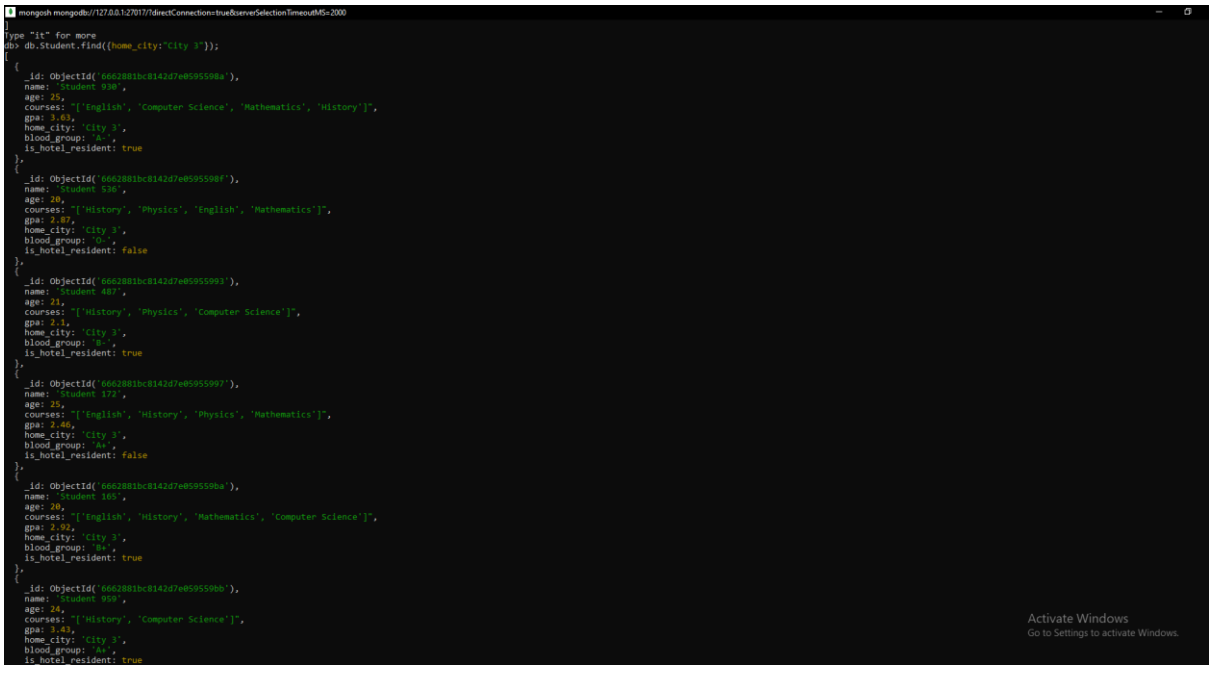
- **Comparison gt and lt**
- **And operator**
- **Or operator**

Comparision operator:

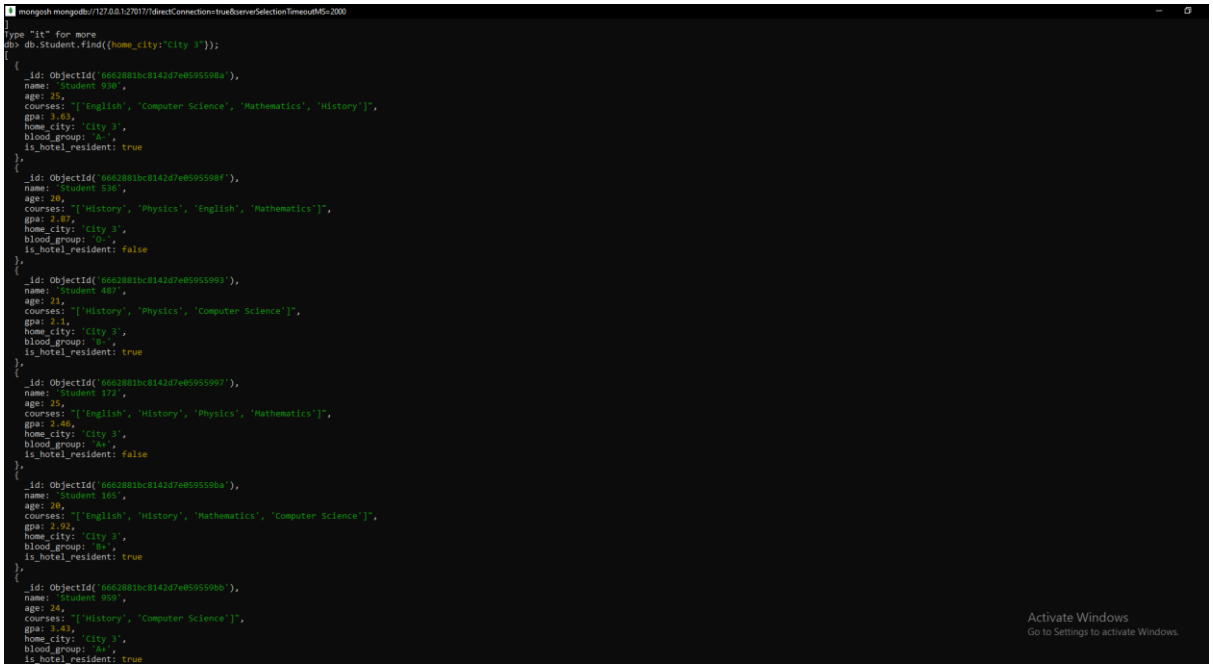
the gt and lt selectors are used in query operations to compare values and retrieve documents that have fields with values greater than (gt) or less than (lt) a specified value. These selectors are commonly used when querying for numeric or date/time values.

```
// Example: Retrieving documents with "age" greater than 25
db.users.find({ age: { $gt: 25 } })

// Example: Retrieving documents with "salary" less than 50000
db.employees.find({ salary: { $lt: 50000 } })

// Example: Retrieving documents with "date" greater than a specific date
db.events.find({ date: { $gt: new Date("2024-06-01") } })
```

And operator:



Or operator :


mongosh mongodb://127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000
Type "it" for more
db> db.Student.find({home_city:"City 3"});
Activate Windows
Go to Settings to activate Windows.