UNIT -II

Using MongoDB Shell

Using MongoDB Shell: Basic Querying, Create and Insert, Explicitly Creating Collections, Inserting Documents Using Loop, Inserting by Explicitly Specifying _id, Update, Delete, Read, Using Indexes, Stepping Beyond the Basics, Using Conditional Operators, Regular Expressions, MapReduce, aggregate(), Designing an Application's Data Model, Relational Data Modeling and Normalization, MongoDB Document Data Model Approach

PYQ : (Previous Year Mumbai University Question)

Nov – 18

1.

a. Consider a Collection users containing the following fields

```
{
id: ObjectID(),
FName: "First Name",
LName: "Last Name",
Age: 30,
Gender: "M",
Country: "Country"
}
```

Where Gender value can be either "M" or "F" or "Other".
Country can be either "UK" or "India" or "USA".
Based on above information write the **MongoDB** query for the following.

i. Update the country to UK for all female users.
ii. Add the new field company to all the documents.
iii. Delete all the documents where Gender = 'M'.
iv. Find out a count of female users who stay in either India or USA.
v. Display the first name and age of all female employees.

2. Write the MongoDB command to create the following with an example

i) Database ii) Collection iii) Document iv) Drop Collection v) Drop Database vi) Index

3. List and Explain the different conditional operators in MongoDB

Apr – 19

1. Discuss Indexes and Its types.
2. Explain the concept of inserting explicitly Specifying _id.

Nov – 19

1. Explain with an example the process of deleting documents in a collection.
2. Differentiate between Single Key and Compound Index.

Nov – 22

1. How can you create a collection explicitly? Explain about selector and projector with example.
2. What is the use of findOne() method? Briefly explain about explain() function.

Sep – 23

b. Consider a MongoDB database that has "movies" collection:

```
{
    id: ObjectId("573a1390f29313caabcd42e8"),
plot: 'A group of bandits stage a brazen train hold-up, only to find a determined posse
hot on their heels.',
genres: [ 'Short', 'Western' ],
runtime: 11,
cast: [
    'A.C. Abadie',
    "Gilbert M. 'Broncho Billy' Anderson",
    'George Barnes',
    'Justus D. Barnes'
  ],
title: 'The Great Train Robbery',
languages: [ 'English' ],
released: ISODate("1903-12-01T00:00:00.000Z"),
directors: [ 'Edwin S. Porter' ], rated: 'TV-G', awards: { wins: 1, nominations: 0, text: '1
win.' }, lastupdated: '2015-08-13 00:27:59.177000000', year: 1903,
imdb: { rating: 7.4, votes: 9847, id: 439 },
countries: [ 'USA' ],
type: 'movie',
tomatoes: {
viewer: { rating: 3.7, numReviews: 2559, meter: 75 },
fresh: 6,
critic: { rating: 7.6, numReviews: 6, meter: 100 },
rotten: 0,
lastUpdated: ISODate("2015-08-08T19:16:10.000Z")
  }
```

Write queries for the following:

i) Find all movies with full information from the 'movies' collection that released in the year 1893.

ii)Find all movies with full information from the 'movies' collection that have a runtime greater than 120 minutes.

iii) Find all movies with title, languages, released, directors, writers, awards, year, genres, runtime, cast, countries from the 'movies' collection in MongoDB that have at least one nomination.

iv) Retrieve all movies with title, languages, released, directors, writers, countries from the 'movies' collection in MongoDB that have a word "scene" in the title.

v) Find all movies with title, languages, released, runtime, directors, writers, countries from the 'movies' collection in MongoDB that have a runtime between 60 and 90 minutes.

1. Illustrate the use of Query document in MongoDB
2. Explain the types of indexes in MongoDB

Nov – 23

1. Explain the selector and projector in the query documents of MongoDB with example
2. Explain MapReduce framework of MongoDB with example.

"Mongo shell comes with the standard distribution of MongoDB. It offers a environment with complete access to the language and the standard functions. It provides a full interface for the MongoDB database."

The MongoDB shell introduction has been divided into three parts in order to make it easier for the readers to grasp and practice the concepts. The first section covers the basic features of the database, including the basic CRUD operators. The next section covers advanced querying. The last section of the chapter explains the two ways of storing and retrieving data: embedding and referencing.

Table 6-1. *SQL and MongoDB Terminology*

SQL	MongoDB
Database	Database
Table	Collection
Row	Document
Column	Field
Index	Index
Joins within table	Embedding and referencing
Primary Key: A column or set of columns can be specified	Primary Key: Automatically set to _id field

Basic Querying:

This section will briefly discuss the CRUD operations (Create, Read, Update, and Delete). Using basic examples and exercises, you will learn how these operations are performed in MongoDB. Also, you will understand how queries are executed in MongoDB.

In contrast to traditional SQL, which is used for querying, MongoDB uses its own JSON-like query language to retrieve information from the stored data.

Installation of MongoDB:

Step 1: Go to the MongoDB Download Center to download the MongoDB Community Server.

Here, You can select any version, Windows, and package according to your requirement. For Windows, we need to choose:

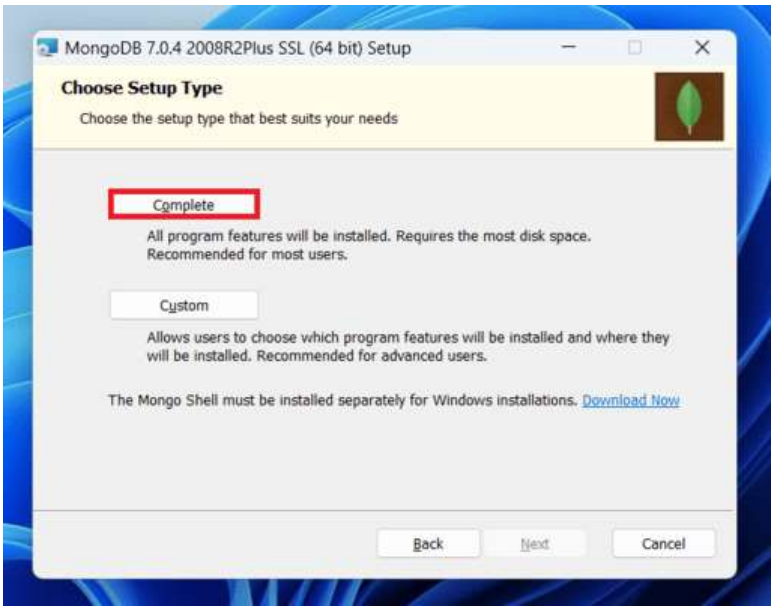
- Version: 7.0.4
- OS: Windows x64
- Package: msi

- **Step 2:** When the download is complete open the msi file and click the *next button* in the startup screen:

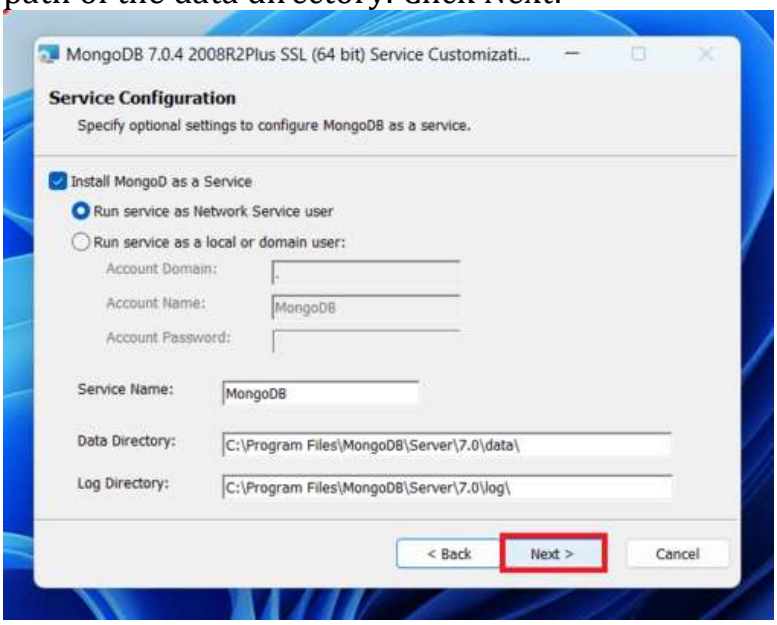
- **Step 3:** Now accept the End-User License Agreement and click the next button:

- **Step 4:** Now select the *complete option* to install all the program features. Here, if you can want to install only selected program

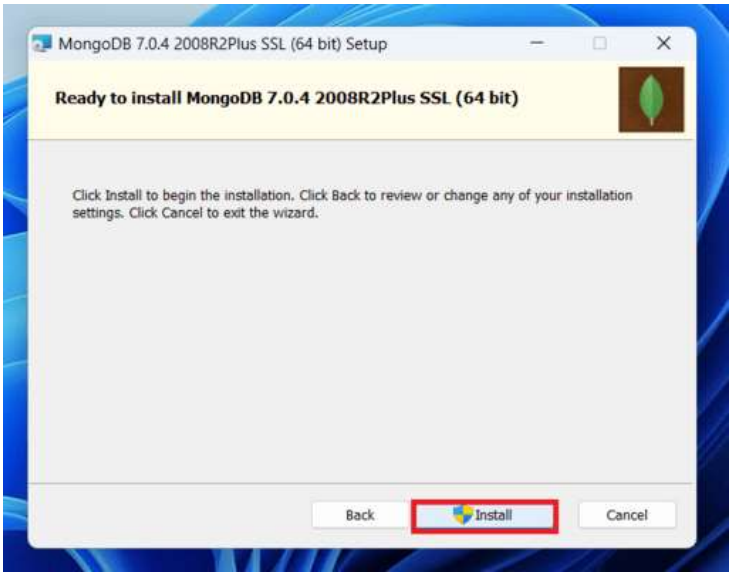
features and want to select the location of the installation, then use the *Custom option:*



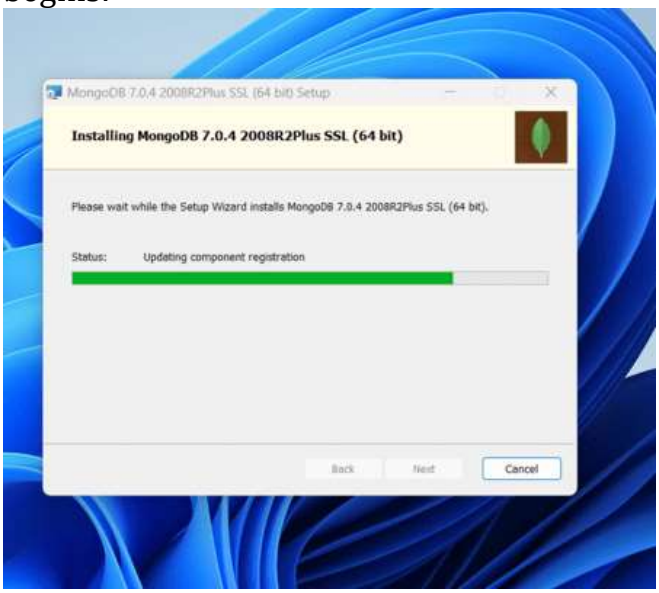
- **Step 5:** Select "Run service as Network Service user" and copy the path of the data directory. Click Next:



- **Step 6:** Click the *Install button* to start the MongoDB installation process:

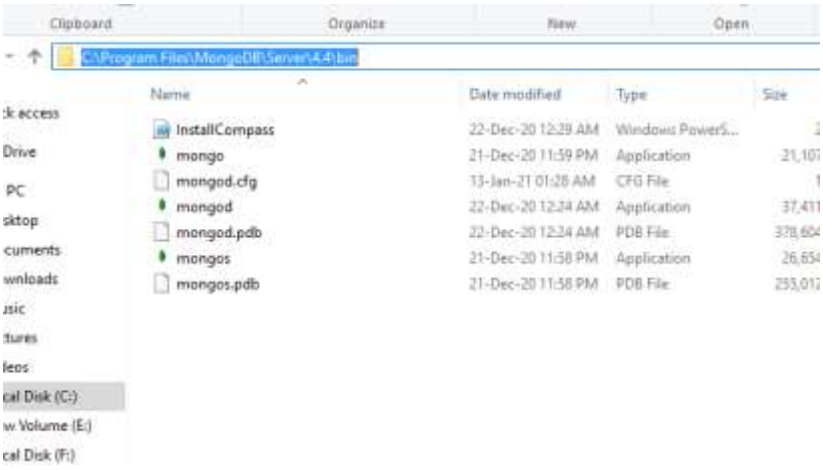


- **Step 7:** After clicking on the install button installation of MongoDB begins:

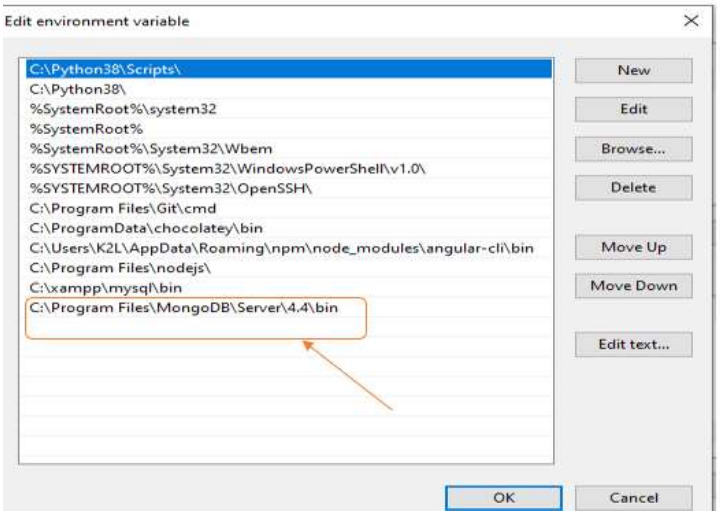


- **Step 8:** Now click the ***Finish button*** to complete the MongoDB installation process:

- **Step 9:** Now we go to the location where MongoDB installed in step 5 in your system and copy the bin path:



- **Step 10:** Now, to create an environment variable open system properties >> Environment Variable >> System variable >> path >> Edit Environment variable and paste the copied link to your environment system and click Ok:



- **Step 11:** After setting the environment variable, we will run the MongoDB server, i.e. mongod. So, open the command prompt and run the following command:

- **mongod**

When you run this command you will get an error i.e. *C:/data/db/ not found*.

- **Step 12:** Now, Open C drive and create a folder named “data” inside this folder create another folder named “db”. After creating these folders. Again open the command prompt and run the following command:

- **mongod**

- Now, this time the MongoDB server(i.e., mongod) will run successfully.

```
C:\Users\NIkhil Chhipa>mongod
{"t":{"$date":"2021-01-31T00:56:54.081+05:30"},"s":"I",  "c":"CONTROL",  "id":23285,   "ctx"
ify --sslDisabledProtocols 'none'"}
{"t":{"$date":"2021-01-31T00:56:54.087+05:30"},"s":"W",  "c":"ASIO",     "id":22601,   "ctx"
}
{"t":{"$date":"2021-01-31T00:56:54.088+05:30"},"s":"I",  "c":"NETWORK",  "id":4648602, "ctx"
{"t":{"$date":"2021-01-31T00:56:54.090+05:30"},"s":"I",  "c":"STORAGE",  "id":4615611, "ctx"
bPath":"C:/data/db/","architecture":"64-bit","host":"DESKTOP-L9MUQ7N"}}
{"t":{"$date":"2021-01-31T00:56:54.090+05:30"},"s":"I",  "c":"CONTROL",  "id":23398,   "ctx"
rgetMinOS":"Windows 7/Windows Server 2008 R2"}}
{"t":{"$date":"2021-01-31T00:56:54.090+05:30"},"s":"I",  "c":"CONTROL",  "id":23403,   "ctx"
gitVersion":"913d6b62acfbb344dde1b116f4161360acd8fd13","modules":[],"allocator":"tcmalloc","
}}}}
{"t":{"$date":"2021-01-31T00:56:54.090+05:30"},"s":"I",  "c":"CONTROL",  "id":51765,   "ctx"
ndows 10","version":"10.0 (build 14393)"}}}
{"t":{"$date":"2021-01-31T00:56:54.090+05:30"},"s":"I",  "c":"CONTROL",  "id":21951,   "ctx"
{"t":{"$date":"2021-01-31T00:56:54.157+05:30"},"s":"I",  "c":"STORAGE",  "id":22270,   "ctx"
:{"dbpath":"C:/data/db/","storageEngine":"wiredTiger"}}
{"t":{"$date":"2021-01-31T00:56:54.158+05:30"},"s":"I",  "c":"STORAGE",  "id":22315,   "ctx"
ize=1491M,session_max=33000,eviction=(threads_min=4,threads_max=4),config_base=false,statist
le_manager=(close_idle_time=100000,close_scan_interval=10,close_handle_minimum=250),statisti
ess],"}}
{"t":{"$date":"2021-01-31T00:56:54.395+05:30"},"s":"I",  "c":"STORAGE",  "id":22430,   "ctx"
95788][3708:140713908197088], txn-recover: [WT_VERB_RECOVERY_PROGRESS] Recovering log 20 thr
{"t":{"$date":"2021-01-31T00:56:54.631+05:30"},"s":"I",  "c":"STORAGE",  "id":22430,   "ctx"
```

--

Show Databases :

Question 1 : Describe how to show all databases in mongodb ? Explain with example

- MongoDB is a NoSQL database that stores data in flexible, JSON-like documents.
- To manage and interact with databases in MongoDB, various commands are used.
- This is often necessary for database administrators and developers to understand the available databases, manage them, and perform operations.

Syntax : show dbs Example : show dbs

To show all databases in MongoDB, you use the `show dbs` command. This command lists all databases along with their sizes

Output:

admin 0.000GB
config 0.000GB
local 0.000GB
mydb 0.002GB

-

Creating Database :

Question 1 : Explain how to create the database or change the database in mongodb with example .
Question 2 : Explain how a new database is created and managed in MongoDB. Provide examples to illustrate the process.
Question 3 : Discuss the steps involved in switching between databases in MongoDB. Include examples to support your explanation.
Question 4 : Illustrate the process of database creation in MongoDB using the use command. How does MongoDB handle database creation implicitly?
Question 5 : Describe how to create a collection in a new database in MongoDB. Explain the role of the use command in this process with examples.
Question 6 : In MongoDB, how can you create and switch to a new database? Provide a detailed explanation with appropriate code snippets.

- In MongoDB, a database is created automatically when you insert some data into it.
- There is no explicit command to create a database. Instead, you use the `use` command to switch to the database you want to create, and then perform an operation that writes data to it.

Create a Database:

1. **Switch to the Database:**
Use the `use` command to switch to the desired database. If the database does not exist, MongoDB will create it when you insert data.

Syntax:
use <databasename>
Example :
use shop

2. **Insert Data into the Database:**
Perform an insert operation to create the collection and add documents.

Syntax :
db.createCollection.insertOne({fieldname:value,fieldname:value,....})
Example :
db.customer.insertOne({ name: "John", age: 30, city: "New York" })

Here, “shop” is the database, `customer` is the collection, and the document being inserted contains fields `name`, `age`, and `city`.

Changing the Database

To change the database, simply use the `use` command to switch to another database.

Syntax :
use <database name>
Example :
use shop (where shop database is already exist)

After executing this command, all subsequent operations will be performed on `shop`.

Question 1 : How can you create a collection explicitly?
Question 2 : How can you create a collection implicitly?
Question 3 : Explain the concept of inserting explicitly Specifying _id

Creating Collection :

In MongoDB, a collection is a group of documents. It is similar to a table in relational databases.

Key Points:

1. Storage of Documents:

   I. A collection stores documents in BSON format (Binary JSON).
   II. Each document is a set of key-value pairs.

2. Dynamic Schema:

I. Unlike traditional databases, collections in MongoDB do not enforce a schema.
II. Documents in a collection can have different fields.

3. Implicit and Explicit Creation:

I. A collection can be created implicitly by inserting a document into a non-existent collection.
II. A collection can also be created explicitly using the createCollection method.

4. No Fixed Structure:

I. Collections do not require a predefined structure.
II. You can add different types of documents to the same collection.

A) Implictly create collection :

Details:

A collection in MongoDB can be created implicitly by inserting a document into a non-existent collection. MongoDB will automatically create the collection when you insert the document.

Syntax:
db.collectionName.insertOne(document)

Example:

1. **Switch to the Desired Database:**
   use schoolDB

2. **Insert a Document to Create the Collection:**
   db.students.insertOne({ name: "Alice", age: 20, major: "Computer Science" })

3. **Verify the Collection Creation:**
   show collections

   Output:
   students

4. **Verify the Document Insertion:**

db.students.find()

Output:
json

```
{ "_id": ObjectId("60b7e4f7d13f1a3d6c8b4567"), "name": "Alice", "age": 20, "major": "Computer Science" }
```

B) Explicit create collection :

Details:

A collection can be explicitly created using the createCollection method. This method allows you to specify options such as validation rules, storage engine options, and more.

Syntax:

db.createCollection("collectionName", options)

Example:

1. **Switch to the Desired Database:**

use schoolDB

2. **Create the Collection Explicitly:**
   db.createCollection("students")

3. **Verify the Collection Creation:**
   show collections

Output:
students

4. **Insert a Document into the Collection:**

db.students.insertOne({ name: "Alice", age: 20, major: "Computer Science" })

5. **Verify the Document Insertion:**
   db.students.find()

Output:

```
{ "_id": ObjectId("60b7e4f7d13f1a3d6c8b4567"), "name": "Alice",
"age": 20, "major": "Computer Science" }
```

Inserting data in collection :

Question 1 : Explain to insert one data at a time or many record at a time with example.
Question 2 : Explain to insert data in collection with example .

To insert data into a MongoDB collection, you can use the `insertOne` method to insert a single document or the `insertMany` method to insert multiple documents at once.

- **Inserting a one record at a time :**

  The `insertOne` method is used to insert a single document into a collection. It is useful when you need to add individual records to the database.

  Syntax :
  db.collectionName.insertOne(document)

  Example :
  db.students.insertOne({ name: "Alice", age: 20, major: "Computer Science" })

  Example explanation :

- Use school
  This command switches the context to the `school` database. If it doesn't exist, MongoDB will create it.
- db.students.insertOne({ name: "Alice", age: 20, major: "Computer Science" })
  This command inserts a single document into the `students` collection with the fields `name`, `age`, and `major`.

- **Inserting Many Records at a Time :**

The `insertMany` method allows you to insert multiple documents into a collection in a single operation. This method is efficient for adding a large number of documents at once.

Syntax :
db.collectionName.insertMany([document1, document2, ...])

Example :

```
db.students.insertMany([
  { name: "Bob", age: 22, major: "Mathematics" },
  { name: "Carol", age: 21, major: "Physics" },
  { name: "David", age: 23, major: "Chemistry" }
])
```

Example explanation :

- use school
  This command switches the context to the `school` database.
- db.students.insertMany([
  { name: "Bob", age: 22, major: "Mathematics" },
  { name: "Carol", age: 21, major: "Physics" },
  { name: "David", age: 23, major: "Chemistry" }
  ])
  This command inserts three documents into the `students` collection. Each document represents a different student with fields `name`, `age`, and `major`.

Inserting Documents Using Loop :

Question 1 : How to insert data in collection using loop . Explain with example .

- To insert multiple documents into a MongoDB collection using a loop, you can write a script that iterates over a set of data and inserts each document into the collection.
- This approach is useful when you need to programmatically add a large number of documents based on some logic or data set
- Using a loop to insert documents allows you to automate the process of adding multiple records to a collection.

- This can be particularly useful when dealing with large datasets or when the data needs to be dynamically generated or processed before insertion.

Syntax :

```
for (let i = 0; i < numberOfDocuments; i++)
{
 db.collectionName.insertOne
   (
    {
    field1: value1,
    field2: value2,
            ...
   }
            );
}
```

Example :

```
  for (let i = 1; i <= 5; i++)
{
   db.students.insertOne
(
     {
     name: "Student" + i,
     age: 20 + i,
     major: "Major" + i
   }
);
            }
```

Example Explanation:

****Loop Structure:**** The loop runs from `i = 1` to `i = 5`, effectively iterating five times.

****Document Creation:**** During each iteration, a new document is created with fields `name`, `age`, and `major`. The values of these fields are dynamically generated based on the loop index `i`.

- `name`: "Student" + i (e.g., "Student1", "Student2", ...)
- `age`: 20 + i (e.g., 21, 22, ...)
- `major`: "Major" + i (e.g., "Major1", "Major2", ...)

Inserting by Explicitly Specifying _id :

Question 1 : Explain the concept of inserting explicitly Specifying _id.

- In MongoDB, the `_id` field is a unique identifier for documents within a collection.
- Each document must have an `_id` field, which can be implicitly created by MongoDB or explicitly set by the user.
- Here's a detailed explanation of both implicit and explicit `_id` fields, including syntax and examples

❖ Implicit `_id` Field

- When you insert a document into a MongoDB collection without specifying an `_id` field, MongoDB automatically creates one for you.
- This auto-generated `_id` is an ObjectId, a 12-byte identifier that is globally unique.

Syntax

```
db.collection.insertOne({
   "name": "John Doe",
   "age": 30
});
```

Example

```
db.users.insertOne({
   "name": "Alice",
   "age": 25
});
```

Example Explanation

In this example, when you insert the document into the `users` collection, MongoDB automatically adds an `_id` field with a unique ObjectId.

The resulting document might look like this:

```
{
```

```
  "_id": ObjectId("60d5f4928b5c3f5c9d27abfa"),
  "name": "Alice",
  "age": 25
}
```

The `_id` field, `ObjectId("60d5f4928b5c3f5c9d27abfa")`, is generated by MongoDB.

- **Explicit `_id` Field**

You can also explicitly specify the `_id` field when inserting a document. This can be any unique value, not just an ObjectId.

Syntax

```
db.collection.insertOne({
  "_id": value,
  "field1": "value1",
  "field2": "value2"
});
```

Example

```
db.users.insertOne({
  "_id": "user123",
  "name": "Bob",
  "age": 28
});
```

Example Explanation

In this example, the document is inserted into the `users` collection with an explicitly specified `_id` field.

The resulting document looks like this:

```
{
  "_id": "user123",
  "name": "Bob",
  "age": 28
}
```

Here, `_id` is `"user123"`, a value chosen by the user instead of an auto-generated ObjectId.

Points to Remember
- **Uniqueness:** The value of `_id` must be unique within the collection. If you try to insert a document with a duplicate `_id`, MongoDB will throw an error.
- **Type:** The `_id` field can be of any type (string, number, ObjectId, etc.), but it's common to use ObjectId for its uniqueness and timestamp features.
- **Indexing:** The `_id` field is automatically indexed, which ensures fast retrieval of documents by `_id`.

In summary, the `_id` field in MongoDB is a critical part of each document, ensuring uniqueness and efficient data retrieval. Whether to create you allow MongoDB it implicitly or set it explicitly, understanding how to use `_id` effectively is essential for working with MongoDB collections.

-

Update :

updateOne :

The `updateOne` method is used to update a single document that matches a specified filter. If there are multiple documents that match the filter, only the first document found will be updated.

Syntax :

```
db.collection.updateOne(
  { filter},{update operation })
```

Example :

```
db.users.updateOne(
  { "name": "Alice" },
  { $set: { "age": 26 } }
)
```

Example Explanation
- The filter `{ "name": "Alice" }` selects the document where the name is "Alice".
- The update operation `{ $set: { "age": 26 } }` sets the `age` field to 26.
- Only the first document that matches the filter (the one with "Alice") will be updated.

updateMany :

- The `updateMany` method is used to update multiple documents that match a specified filter. All documents that match the filter will be updated.

Syntax :

```
db.collection.updateMany(
 { condition},{update operation })
```

Example :

```
db.users.updateMany(
  { "city": "New York" },
  { $set: { "city": "NYC" } }
)
```

Example Explanation

- The filter `{ "city": "New York" }` selects all documents where the city is "New York".
- The update operation `{ $set: { "city": "NYC" } }` sets the `city` field to "NYC".
- All documents that match the filter (both "Alice" and "David") will have their `city` field updated to "NYC".

Delete :

Question 1 : Explain with an example the process of deleting documents in a collection

In MongoDB, `deleteOne` and `deleteMany` are methods used to remove documents from a collection.

deleteOne :

The `deleteOne` method removes a single document that matches a specified filter. If multiple documents match the filter, only the first one found is deleted.

Syntax :
db.collection.deleteOne(filter, options)
- **filter**: A document that specifies the criteria for deletion.
- **options**: Optional. A document specifying additional options, such as collation.

Example :

Suppose we have a collection `users` with the following documents:

```
[
 { "_id": 1, "name": "Alice", "age": 28 },
 { "_id": 2, "name": "Bob", "age": 34 },
 { "_id": 3, "name": "Charlie", "age": 28 }
]
```

To delete one document where the `age` is 28:

```
db.users.deleteOne({ age: 28 })
```

Example Explanation :
- This command will delete the first document it finds with `age: 28`.
- The remaining documents in the `users` collection might be:

```
[
  { "_id": 2, "name": "Bob", "age": 34 },
  { "_id": 3, "name": "Charlie", "age": 28 }
]
```

deleteMany :

The deleteMany method removes all documents that match a specified filter.

Syntax

db.collection.deleteMany(filter, options)

- **filter**: A document that specifies the criteria for deletion.
- **options**: Optional. A document specifying additional options, such as collation.

Example

Using the same `users` collection, to delete all documents where the `age` is 28:

db.users.deleteMany({ age: 28 })

Example Explanation
- This command will delete all documents with `age: 28`.
- The remaining document in the `users` collection might be:
```
 [
   { "_id": 2, "name": "Bob", "age": 34 }
 ]
```

Detailed Example for `deleteOne` and `deleteMany`

Let's say our `users` collection is as follows:
```
 [
  { "_id": 1, "name": "Alice", "age": 28 },
  { "_id": 2, "name": "Bob", "age": 34 },
  { "_id": 3, "name": "Charlie", "age": 28 },
  { "_id": 4, "name": "David", "age": 40 }
]
```

Using `deleteOne`
db.users.deleteOne({ age: 28 })
- This command will remove the first document where `age` is 28.
- The collection after this operation:
```
 [
   { "_id": 2, "name": "Bob", "age": 34 },
   { "_id": 3, "name": "Charlie", "age": 28 },
   { "_id": 4, "name": "David", "age": 40 }
 ]
```

Using `deleteMany`
db.users.deleteMany({ age: 28 })
- This command will remove all documents where `age` is 28.
- The collection after this operation:

```
 [
   { "_id": 2, "name": "Bob", "age": 34 },
   { "_id": 4, "name": "David", "age": 40 }
```

]

These methods are powerful tools for managing data in MongoDB, allowing precise control over which documents are removed based on specified criteria.

remove :

Note : We can use remove method to delete the data The remove method in MongoDB was used in earlier versions to delete documents from a collection. However, it's been deprecated in favor of deleteOne and deleteMany. Despite this, here's an explanation of remove for historical context.

- The remove method deletes documents from a collection based on a specified filter.
- You can use remove to delete a single document or multiple documents.
- The method can also take an optional parameter to limit the operation to a single document.

Syntax :

```
db.collection.remove(
  <query>,
  {
    justOne: <boolean>,
    writeConcern: <document>
  }
)
```

- **<query>**: The selection criteria for the documents to remove. It specifies the conditions that the documents must meet to be removed.
- **justOne**: Optional. If set to true, only one document matching the query criteria will be removed. The default is false, which means all documents matching the criteria will be removed.
- **writeConcern**: Optional. Specifies the level of acknowledgment requested from MongoDB for write operations.

Example

Let's consider a collection `students` with the following documents:

```
db.students.insertMany([
  { name: "John", age: 25, major: "Computer Science" },
  { name: "Jane", age: 22, major: "Mathematics" },
```

```
  { name: "Jim", age: 25, major: "Physics" },
  { name: "Jack", age: 25, major: "Computer Science" }
]);
```

Remove multiple documents:
db.students.remove({ age: 25 });
This command removes all documents where the `age` field is `25`.

Remove a single document:
db.students.remove({ age: 25 }, { justOne: true });
This command removes only one document where the `age` field is `25`.

Example Explanation

1. **Remove multiple documents:**

```
db.students.remove({ age: 25 });
```

- **Query**: `{ age: 25 }` - This selects all documents where the `age` field is `25`.
- **justOne**: Not specified (default is `false`) - This means all documents matching the query will be removed.

Before removal, the `students` collection contains:

```
[
  { name: "John", age: 25, major: "Computer Science" },
  { name: "Jane", age: 22, major: "Mathematics" },
  { name: "Jim", age: 25, major: "Physics" },
  { name: "Jack", age: 25, major: "Computer Science" }
]
```

After the removal, the collection will be:

```
[
{ name: "Jane", age: 22, major: "Mathematics" }
]
```

2. **Remove a single document:**

```
db.students.remove({ age: 25 }, { justOne: true });
```

- **Query**: `{ age: 25 }` - This selects all documents where the `age` field is `25`.
- **justOne**: `true` - This means only one document matching the query will be removed.

Before removal, the `students` collection contains:

```
[
  { name: "John", age: 25, major: "Computer Science" },
  { name: "Jane", age: 22, major: "Mathematics" },
  { name: "Jim", age: 25, major: "Physics" },
  { name: "Jack", age: 25, major: "Computer Science" }
]
```

After the removal, the collection will have removed just one of the documents with `age` 25:

```
[
  { name: "Jane", age: 22, major: "Mathematics" },
  { name: "Jim", age: 25, major: "Physics" },
  { name: "Jack", age: 25, major: "Computer Science" }
]
```

Note

Even though `remove` is deprecated, understanding it is still useful for working with older MongoDB codebases. For new applications, use the `deleteOne` and `deleteMany` methods, which provide more explicit control over deletion operations.

Read : (Querying Data or Filtering Data)

Question 1 :What is the use of findOne() method? Briefly explain about explain() function.

In MongoDB, the find and findOne methods are used to query documents from a collection.

####find Method

Description

The find method is used to retrieve multiple documents from a collection that match a specified filter. It returns a cursor to the documents, which allows for efficient handling of large datasets.

Syntax :

db.collection.find(query,projection)

- query: A document that specifies the criteria for selecting documents. If omitted, all documents in the collection are returned.
- projection (optional): A document that specifies the fields to return or exclude. If omitted, all fields are returned.

Example :

Consider a collection users with the following documents:

```
[
 { "_id": 1, "name": "Alice", "age": 25, "city": "New York" },
 { "_id": 2, "name": "Bob", "age": 30, "city": "San Francisco" },
 { "_id": 3, "name": "Charlie", "age": 35, "city": "Los Angeles" }
]
```

To find all users aged 30 and above:

```
db.users.find({ age: { $gte: 30 } })
```

##Example Explanation

The query { age: { $gte: 30 } } uses the $gte (greater than or equal) operator to filter documents where the age field is 30 or greater. The result would be:

Output :

```
[
 { "_id": 2, "name": "Bob", "age": 30, "city": "San Francisco" },
 { "_id": 3, "name": "Charlie", "age": 35, "city": "Los Angeles" }
]
```

####findOne :

Description

The findOne method is used to retrieve a single document from a collection that matches a specified filter. If multiple documents match the

filter, it returns the first document found according to the collection's natural order.

Syntax

db.collection.findOne(query, projection)

- query: A document that specifies the criteria for selecting the document. If omitted, the first document in the collection is returned.
- projection (optional): A document that specifies the fields to return or exclude. If omitted, all fields are returned.

Example :

Using the same users collection:

To find a user named "Alice":
db.users.findOne({ name: "Alice" })

Example Explanation

The query { name: "Alice" } filters documents where the name field is "Alice". Since there is only one document with this name, the result would be:
{ "_id": 1, "name": "Alice", "age": 25, "city": "New York" }

Selector :

Question 1 : Explain the selector in mongodb with example .
Question 2 : Explain about selector and projector with example.

- In MongoDB, selectors are used to filter documents based on specified conditions.
- They are used in various methods like `find()`, `update()`, `delete()`, etc., to match documents that meet certain criteria.
- A selector is a document that describes the conditions that the documents must meet to be selected.

Syntax :

{ <field>: <value> }

You can use comparison operators to specify more complex conditions:

{ <field>: { <operator>: <value> } }

Common operators include:

- `$eq`: Equal to
- `$gt`: Greater than
- `$gte`: Greater than or equal to
- `$lt`: Less than
- `$lte`: Less than or equal to
- `$ne`: Not equal to
- `$in`: In array
- `$nin`: Not in array

Example

Let's consider a MongoDB collection named `students` with the following documents:

Output :

```
[
 { "_id": 1, "name": "Alice", "age": 24, "grade": "A" },
 { "_id": 2, "name": "Bob", "age": 22, "grade": "B" },
 { "_id": 3, "name": "Charlie", "age": 23, "grade": "A" },
 { "_id": 4, "name": "David", "age": 25, "grade": "C" }
]
```

Query Example 1: Find students with grade "A"

Selector:

```
{ "grade": "A" }
```

Usage:

```
db.students.find({ "grade": "A" })
```

Result:

```
[
 { "_id": 1, "name": "Alice", "age": 24, "grade": "A" },
 { "_id": 3, "name": "Charlie", "age": 23, "grade": "A" }
]
```

Explanation:
This query uses a selector to match documents where the `grade` field is equal to "A". The `find()` method returns all documents that meet this criterion, which are Alice and Charlie.

Query Example 2: Find students older than 23

Selector:
{ "age": { "$gt": 23 } }

Usage:
db.students.find({ "age": { "$gt": 23 } })

Result:
```
[
 { "_id": 1, "name": "Alice", "age": 24, "grade": "A" },
 { "_id": 4, "name": "David", "age": 25, "grade": "C" }
]
```

Explanation:
This query uses a selector with the `$gt` (greater than) operator to match documents where the `age` field is greater than 23. The `find()` method returns documents for Alice and David, who are 24 and 25 years old, respectively.

Additional Examples
Example 3: Find students with grade "A" and age less than 25

Selector:
{ "grade": "A", "age": { "$lt": 25 } }
Usage:
db.students.find({ "grade": "A", "age": { "$lt": 25 } })

Result:
```
[
 { "_id": 1, "name": "Alice", "age": 24, "grade": "A" },
 { "_id": 3, "name": "Charlie", "age": 23, "grade": "A" }
]
```

Explanation:

This query combines two conditions using a selector. It matches documents where the `grade` field is "A" and the `age` field is less than 25. The `find()` method returns documents for Alice and Charlie.

Projector :

Question 1 : Explain about projector with example.

In MongoDB, a projector is used to control the fields that are returned in query results. By default, a query returns all fields in the matching documents. However, you can use projection to return only specific fields, which can reduce the amount of data transferred and improve query performance. This is particularly useful when dealing with large documents or when you only need a subset of fields.

Syntax :

db.collection.find(query, projection)

- `query`: The selection criteria for the documents.
- `projection`: An object that specifies the fields to include or exclude.

In the projection object, you can use:
- `1` to include a field.
- `0` to exclude a field.

Note: The `_id` field is included by default. To exclude it, you need to explicitly set `_id: 0`.

Example :

Suppose you have a collection named `students` with the following documents:

```
{
 "_id": 1,
 "name": "John Doe",
 "age": 21,
 "course": "Computer Science",
```

```
  "grade": "A"
},
{
  "_id": 2,
  "name": "Jane Smith",
  "age": 22,
  "course": "Information Technology",
  "grade": "B"
}
```

If you want to retrieve only the `name` and `course` fields for all students, you can use the following query:

```
db.students.find({}, { name: 1, course: 1, _id: 0 })
```

Example Explanation :

- **Query**: `{}` - This selects all documents in the `students` collection.
- **Projection**: `{ name: 1, course: 1, _id: 0 }` - This specifies that only the `name` and `course` fields should be included in the output, and the `_id` field should be excluded.

Output :

```
[
  { "name": "John Doe", "course": "Computer Science" },
  { "name": "Jane Smith", "course": "Information Technology" }
]
```

Note : By default _id is present if you have not mentioned it to 0 .

Sort :

- MongoDB provides the `sort()` method to arrange the documents in a particular order.
  - Sorting can be done in ascending or descending order. This method is crucial for organizing query results and is often used to improve readability and data analysis.

Explanation :

- Sorting is the process of ordering data in a specific sequence. In MongoDB, the `sort()` method is used to specify the order in which documents are returned.
- It takes an object as a parameter where the field to be sorted is the key and the sorting order is the value. The sorting order can be `1` for ascending or `-1` for descending.

Syntax :
db.collection.find(query).sort({ field1: order, field2: order, ... })

- `query`: The selection criteria for the documents to be returned (optional).
- `field1`, `field2`, ...: The fields on which to sort.
- `order`: The sorting order (`1` for ascending, `-1` for descending).

Example :

Consider a collection `students` with the following documents:

{ "_id": 1, "name": "Alice", "age": 24, "score": 85 }
{ "_id": 2, "name": "Bob", "age": 22, "score": 90 }
{ "_id": 3, "name": "Charlie", "age": 23, "score": 80 }
{ "_id": 4, "name": "David", "age": 22, "score": 92 }
To sort the documents by `age` in ascending order and then by `score` in descending order, use the following query:

db.students.find().sort({ age: 1, score: -1 })

Example Explanation

1. **Query Execution**:
   - `db.students.find()`: This retrieves all the documents from the `students` collection.

2. **Sorting Specification**:
   - `.sort({ age: 1, score: -1 })`:
     - `{ age: 1 }`: This part of the object specifies that the documents should be sorted by the `age` field in ascending order.
     - `{ score: -1 }`: This part of the object specifies that if multiple documents have the same `age`, they should be further sorted by the `score` field in descending order.

3. **Result**:
   - The documents will be sorted first by `age` and, for those with the same `age`, by `score` in descending order:

```
{ "_id": 2, "name": "Bob", "age": 22, "score": 90 }
{ "_id": 4, "name": "David", "age": 22, "score": 92 }
{ "_id": 3, "name": "Charlie", "age": 23, "score": 80 }
{ "_id": 1, "name": "Alice", "age": 24, "score": 85 }
```

In summary, the `sort()` method in MongoDB is a powerful tool for arranging query results in a specific order, making it easier to analyze and present data. The syntax is straightforward, and multiple fields can be used to achieve complex sorting requirements.

---

limit () :

- In MongoDB, the `limit` method is used to restrict the number of documents returned in a query result.
- This is particularly useful when you need to fetch only a subset of the data, such as the top results, for performance optimization, or when implementing pagination

Syntax :

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

- **query**: Specifies the selection criteria using query operators.
- **number**: The maximum number of documents to return.

Example :

Suppose we have a collection called `students` with the following documents:

```
[
 { "_id": 1, "name": "Alice", "age": 22 },
 { "_id": 2, "name": "Bob", "age": 23 },
 { "_id": 3, "name": "Charlie", "age": 24 },
 { "_id": 4, "name": "David", "age": 25 },
```

{ "_id": 5, "name": "Eva", "age": 26 }
]
If you want to retrieve only the first 3 documents from this collection, you can use the `limit` method as follows:
db.students.find({}).limit(3)

Example Explanation
1. **Collection**: The `students` collection contains 5 documents.
2. **Query**: `{}` (empty query) means we are selecting all documents.
3. **Limit**: `3` specifies that we want only the first 3 documents

The result of this query will be:
[
{ "_id": 1, "name": "Alice", "age": 22 },
{ "_id": 2, "name": "Bob", "age": 23 },
{ "_id": 3, "name": "Charlie", "age": 24 }
]

By using the `limit` method, MongoDB restricts the output to the first 3 documents from the collection. This is useful for managing large datasets and improving query performance by avoiding the retrieval of unnecessary data.

Skip :

The `skip()` method in MongoDB is used to skip a specified number of documents in a query result set. This is particularly useful for implementing pagination in your application, where you might want to retrieve a subset of documents starting from a specific point.

Syntax :

db.collection.find(query).skip(number)

- `query`: The query criteria to filter the documents.
- `number`: The number of documents to skip from the beginning of the result set.

Example :

Suppose you have a collection `students` with the following documents:

```
[
  { "_id": 1, "name": "Alice", "age": 21 },
  { "_id": 2, "name": "Bob", "age": 22 },
  { "_id": 3, "name": "Charlie", "age": 23 },
  { "_id": 4, "name": "David", "age": 24 },
  { "_id": 5, "name": "Eve", "age": 25 }
]
```

To skip the first 2 documents and retrieve the rest, you can use the following query:

```
db.students.find().skip(2)
```

Example Explanation :
When the `skip(2)` method is used with `find()`, MongoDB will skip the first 2 documents in the `students` collection and return the remaining documents. The result of the above query would be:

```
[
  { "_id": 3, "name": "Charlie", "age": 23 },
  { "_id": 4, "name": "David", "age": 24 },
  { "_id": 5, "name": "Eve", "age": 25 }
]
```

In this example, the documents with `_id` 1 and 2 are skipped, and the documents with `_id` 3, 4, and 5 are returned.

---

findOne :

The `findOne` method in MongoDB is used to retrieve a single document from a collection that matches the given query criteria. If multiple documents match the criteria, it returns the first one it encounters based on the natural order of the documents in the collection.

Syntax

```
db.collection.findOne(query, projection)
```

- `query`: Specifies the selection criteria using a query object. If omitted, it defaults to an empty query which matches all documents.
- `projection` (optional): Specifies the fields to include or exclude in the returned document. It is a document where the keys are the field names, and the values are `1` to include or `0` to exclude the fields.

Example

Let's say we have a collection named `students` with the following documents:

```
[
 { "_id": 1, "name": "Alice", "age": 21, "grade": "A" },
 { "_id": 2, "name": "Bob", "age": 22, "grade": "B" },
 { "_id": 3, "name": "Charlie", "age": 23, "grade": "C" }
]
```

To find a single document where the `name` is "Bob", we can use the `findOne` method as follows:

```
db.students.findOne({ name: "Bob" })
```

Output :

```
{ "_id": 2, "name": "Bob", "age": 22, "grade": "B" }
```

Example Explanation

1. **Query**: The query `{ name: "Bob" }` specifies the selection criteria to find documents where the `name` field is "Bob".
2. **Return**: The method returns the first document that matches the query. In this case, it is the document with `_id: 2`.
3. **Result**: The result includes all fields of the matched document by default, unless a projection is specified.

Using Cursor :

Explanation:

- In MongoDB, a cursor is a pointer to the result set of a query.
- When you perform a query operation, MongoDB returns a cursor, which allows you to iterate over the documents in the result set one by one.
- Cursors are used to manage large amounts of data efficiently by retrieving results in batches, rather than all at once.

Syntax:

The basic syntax to obtain a cursor in MongoDB is:

```
var cursor = db.collectionname.find(query,projection)
```

Here,

- query specifies the selection criteria
- projection specifies the fields to include or exclude in the returned documents.

Example :

Consider a collection named students with documents containing student information.

1. Inserting Documents:

```
db.students.insertMany([
  { name: "Alice", age: 22, grade: "A" },
  { name: "Bob", age: 23, grade: "B" },
  { name: "Charlie", age: 21, grade: "A" }
]);
```

In this step, three student documents are inserted into the students collection.

2. Creating a Cursor:

```
var cursor = db.students.find({ grade: "A" });
```

Here, we perform a find query on the students collection to retrieve all documents where the grade field is "A". This query returns a cursor that points to the result set.

3. Manipulate using while loop :

```
while(cursor.hasNext())
{
        printjson(c.next())
}
```

Output :

```
{ name: "Alice", age: 22, grade: "A" },
{ name: "Charlie", age: 21, grade: "A" }
```

4. Accessing using index :
   The variable to which the cursor object is assigned can also be manipulated as an array

```
printjson(c[1])
```

Output :

```
{ name: "Charlie", age: 21, grade: "A" }
```

- The next() function returns the next document.
- The hasNext() function returns true if a document exists
- printjson() renders the output in JSON format

-

explain () :

- The `explain` method in MongoDB provides detailed information about how a query is executed.
- This method is used to analyze and understand the performance characteristics of a query by showing the execution plan that MongoDB uses.
- The `explain` method can be particularly useful for optimizing queries, as it reveals details like the use of indexes, the number of documents scanned, and the stages of the query execution.

Syntax :

```
db.collection.find(query).explain(verbosity)
```

- **query**: The query object to match documents in the collection.

- **verbosity**: Optional parameter that specifies the level of detail in the output. It can take one of three values:
  - `"queryPlanner"`: Provides information about the query planner's choice of plan.
  - `"executionStats"`: Provides detailed execution statistics along with the query plan.
  - `"allPlansExecution"`: Provides information about all query plans considered by the query planner.

Example
Let's consider a collection `students` with documents containing information about students, such as name and age.

```
db.students.find({ age: { $gt: 20 } }).explain("executionStats")
```

Example Explanation
In this example, we have a `students` collection, and we want to find all students whose age is greater than 20. The `explain` method is called with the `executionStats` verbosity level to provide detailed statistics about the execution of the query.

The output of the `explain` method will be a detailed document that includes the following key information:

1. **Query Planner Information**:
   - **winningPlan**: The plan that the query planner chose to execute.
   - **rejectedPlans**: Other plans considered but not chosen by the query planner.

2. **Execution Stats**:
   - **nReturned**: The number of documents returned by the query.
   - **executionTimeMillis**: The time taken to execute the query in milliseconds.
   - **totalKeysExamined**: The number of index keys examined.
   - **totalDocsExamined**: The number of documents examined.

3. **Index Information** (if applicable):
   - Details about the indexes used in the query, if any, and their performance impact.

Sample Output:

```json
{
 "queryPlanner": {
  "plannerVersion": 1,
  "namespace": "database.students",
  "indexFilterSet": false,
  "parsedQuery": {
   "age": {
    "$gt": 20
   }
  },
  "winningPlan": {
   "stage": "COLLSCAN",
   "direction": "forward"
  },
  "rejectedPlans": []
 },
 "executionStats": {
  "executionSuccess": true,
  "nReturned": 10,
  "executionTimeMillis": 2,
  "totalKeysExamined": 0,
  "totalDocsExamined": 100,
  "executionStages": {
   "stage": "COLLSCAN",
   "nReturned": 10,
   "executionTimeMillisEstimate": 1,
   "works": 102,
   "advanced": 10,
   "needTime": 91,
   "needYield": 0,
   "saveState": 0,
   "restoreState": 0,
   "isEOF": 1,
   "direction": "forward",
   "docsExamined": 100
  }
 },
 "serverInfo": {
  "host": "hostname",
  "port": 27017,
  "version": "4.4.0",
```

```
    "gitVersion": "b4042c0b68f944b69e32b0f4b6271f6d4f305f51"
  }
}
```

Using Index :

Question 1 : Discuss Indexes and Its types.
Question 2 : Differentiate between Single Key and Compound Index.

In MongoDB, indexing is crucial for optimizing query performance by facilitating faster data retrieval. Here's an explanation, syntax, example, and example explanation for the different aspects of indexing:

Index Types in MongoDB:

- **Single Index Key**: Indexes created on a single field.
- **Compound Index**: Indexes created on multiple fields.
- **Unique Index**: Ensures the indexed fields have unique values across the collection.

How Indexing Works:

Indexes store a small portion of the collection's data in an easy-to-traverse form. This allows MongoDB to quickly locate documents without scanning every document in a collection.

1. Single Index Key:

Explanation: Single index key indexes are created on a single field in a MongoDB collection. They speed up queries that filter or sort by the indexed field.

Syntax:

```
db.collection.createIndex({ field: 1 })
```

- { field: 1 }: Indicates indexing on the field in ascending order (-1 for descending).

Example:

```
db.users.createIndex({ username: 1 })
```

Example Explanation: Creates an index on the username field in the users collection. This index optimizes queries that involve filtering or sorting by username, enhancing query performance.

2. Compound Index:

Explanation: Compound indexes are created on multiple fields in a MongoDB collection. They support queries that filter or sort by combinations of these fields.

Syntax:

```
db.collection.createIndex({ field1: 1, field2: -1 })
```

- { field1: 1, field2: -1 }: Indexes field1 in ascending order and field2 in descending order (-1).

Example:

```
db.orders.createIndex({ customer_id: 1, order_date: -1 })
```

Example Explanation: Creates a compound index on customer_id (ascending) and order_date (descending) in the orders collection. This index optimizes queries that filter or sort orders by customer and date, improving query performance.

3. Unique Index:

Explanation: Unique indexes ensure that the indexed fields contain unique values across the collection. They enforce uniqueness constraints.

Syntax:

```
db.collection.createIndex({ field: 1 }, { unique: true })
```

- { unique: true }: Specifies uniqueness constraint on the indexed field (false for non-unique).

Example:

db.products.createIndex({ product_code: 1 }, { unique: true })

Example Explanation: Creates a unique index on product_code in the products collection. This ensures each product_code value is unique, preventing duplicate entries.

System.indexes :

Explanation: system.indexes is a MongoDB collection that stores information about all indexes in a database. It's used internally by MongoDB to manage and query indexes.

Example:

db.system.indexes.find()

Example Explanation: Executes a query to retrieve information about all indexes in the current database (db). It returns details such as index names, keys, and options.

Drop Index :

Explanation: Drop index removes an existing index from a collection in MongoDB. It's useful for index maintenance and optimization.

Syntax:
db.collection.dropIndex("index_name")

- "index_name": Name of the index to be dropped.

Example:
db.users.dropIndex("username_1")

Example Explanation: Removes the index named username_1 from the users collection. This operation frees up resources and space used by the index.

Re Index :

Explanation: Reindexing rebuilds all indexes on a collection in MongoDB. It can resolve index fragmentation and optimize index storage.

Syntax:
javascript

db.collection.reIndex()

Example:
javascript

db.orders.reIndex()

Example Explanation: Rebuilds all indexes on the orders collection. This operation ensures that indexes are optimized and up-to-date, maintaining efficient query performance.

Differentiate between Single Key Index and Compound Key Index:

Points	Single Key Index	Compound Index
Definition	Index on a single field of a document.	Index that combines multiple fields into a single index key
Structure:	Contains entries pointing to values within a single field	Concatenates fields into a combined index key.
Usage:	Efficient for queries on a single field.	Ideal for queries involving multiple fields simultaneously
Example:	Index on name field in a users collection.	Index on { lastName: 1, firstName: 1 } in a contacts collection.
Query Optimization	Optimizes queries filtering or sorting by a single field	: Optimizes queries filtering, sorting, or matching on multiple fields
Flexibility:	Limited to optimizing queries on a single field.	Offers flexibility for queries involving combinations of fields

Performance Impact:	Benefits single-field queries; less effective for complex queries.	Improves performance for queries on indexed fields and combinations thereof.
Index Size	Generally smaller compared to compound indexes.	Larger due to combined index key size.
Maintenance:	Easier to maintain as it focuses on a single field.	Requires careful consideration of field combinations for optimal performance.
Decision Criteria:	Choose for straightforward queries on individual fields.	Choose when optimizing for queries involving multiple fields or complex conditions.

Stepping beyond the basics :

This section will cover advanced querying using conditional operators and regular expressions in the selector part. Each of these operators and regular expressions provides you with more control over the queries you write and consequently over the information you can fetch from the MongoDB database.

Using conditional operator :

Question 1 : List and Explain the different conditional operators in MongoDB

Conditional operators enable you to have more control over the data you are trying to extract from the database.

1. $lt : less than
2. $lte : less than or equal to
3. $gt : greater than
4. $gte : greater than or equal to
5. $in : In
6. $nin : Not in
7. $not : negative of value

8. $eq : equal to
9. $neq : not equal to

1. $lt (Less Than) :

- **Explanation:** Selects documents where the value of the field is less than the specified value.
- **Syntax:** { field: { $lt: value } }
- **Example:** Find all documents where the "score" field is less than 50.
  db.scores.find({ score: { $lt: 50 } })

- **Example Explanation:** This query will return documents where the "score" field contains values strictly less than 50.

2. $lte (Less Than or Equal) :

- **Explanation:** Selects documents where the value of the field is less than or equal to the specified value.
- **Syntax:** { field: { $lte: value } }
- **Example:** Find all documents where the "quantity" field is less than or equal to 100.
  db.inventory.find({ quantity: { $lte: 100 } })

- **Example Explanation:** This query will return documents where the "quantity" field contains values less than or equal to 100.

3. $gt (Greater Than) :

- **Explanation:** Selects documents where the value of the field is greater than the specified value.
- **Syntax:** { field: { $gt: value } }
- **Example:** Find all documents where the "price" field is greater than 10.
  mongodb

  db.products.find({ price: { $gt: 10 } })

- **Example Explanation:** This query will return documents where the "price" field contains values strictly greater than 10.

4. $gte (Greater Than or Equal) :

- **Explanation:** Selects documents where the value of the field is greater than or equal to the specified value.
- **Syntax:** { field: { $gte: value } }
- **Example:** Find all documents where the "age" field is greater than or equal to 18.
  mongodb

  db.users.find({ age: { $gte: 18 } })

- **Example Explanation:** This query will return documents where the "age" field contains values greater than or equal to 18.

5. $in (In) :

- **Explanation:** Selects documents where the value of a field equals any value in the specified array.
- **Syntax:** { field: { $in: [value1, value2, ...] } }
- **Example:** Find all documents where the "status" field is either "available" or "pending".
  db.orders.find({ status: { $in: ["available", "pending"] } })

- **Example Explanation:** This query will return documents where the "status" field matches either "available" or "pending".

6. $nin (Not In)

- **Explanation:** Selects documents where the value of a field does not equal any value in the specified array.
- **Syntax:** { field: { $nin: [value1, value2, ...] } }
- **Example:** Find all documents where the "type" field is neither "fruit" nor "vegetable".
  mongodb

  db.products.find({ type: { $nin: ["fruit", "vegetable"] } })

- **Example Explanation:** This query will return documents where the "type" field does not match "fruit" or "vegetable".

7. $not (Not) :

- **Explanation:** Selects documents where the value of a field does not match the specified value.
- **Syntax:** { field: { $not: { $eq: value } } }
- **Example:** Find all documents where the "status" field is not "completed".
  mongodb

  db.tasks.find({ status: { $not: { $eq: "completed" } } })

- **Example Explanation:** This query will return documents where the "status" field is anything other than "completed".

8. $eq (Equal) :

- **Explanation:** Selects documents where the value of the field is equal to the specified value.
- **Syntax:** { field: { $eq: value } }
- **Example:** Find all documents where the "status" field is "active".

db.users.find({ status: { $eq: "active" } })

- **Example Explanation:** This query will return documents where the "status" field contains the value "active".

9. $ne (Not Equal)

- **Explanation:** Selects documents where the value of the field is not equal to the specified value.
- **Syntax:** { field: { $ne: value } }
- **Example:** Find all documents where the "category" field is not "electronics".

db.products.find({ category: { $ne: "electronics" } })

- **Example Explanation:** This query will return documents where the "category" field contains any value other than "electronics".

Regular Expression :

Regular expressions are useful in scenarios where you want to find students with name starting with “A” . In order to understand this, let’s add three or four more students with different names.

> db.students.insert({Name:"Student1", Age:30, Gender:"M", Class: "Biology", Score:90})

> db.students.insert({Name:"Student2", Age:30, Gender:"M", Class: "Chemistry", Score:90})

> db.students.insert({Name:"Test1", Age:30, Gender:"M", Class: "Chemistry", Score:90})

> db.students.insert({Name:"Test2", Age:30, Gender:"M", Class: "Chemistry", Score:90})

> db.students.insert({Name:"Test3", Age:30, Gender:"M", Class: "Chemistry", Score:90})

Say you want to find all students with names starting with “St” or “Te” and whose class begins with “Che”. The same can be filtered using regular expressions, like so:

> db.students.find({"Name":/(St|Te)*/i, "Class":/(Che)/i})

{ "_id" : ObjectId("52f89ecae451bb7a56e59086"), "Name" : "Student2", "Age" : 30, "Gender" : "M", "Class" : "Chemistry", "Score" : 90 }

.....

{ "_id" : ObjectId("52f89f06e451bb7a56e59089"), "Name" : "Test3", "Age" : 30, "Gender" : "M", "Class" : "Chemistry", "Score" : 90 }

> In order to understand how the regular expression works, let's take the query "Name":/(St|Te)*/i .

//i indicates that the regex is case insensitive.

(St|Te)* means the Name string must start with either "St" or "Te".

The * at the end means it will match anything after that.

When you put everything together, you are doing a case insensitive match of names that have either "St" or "Te" at the beginning of them. In the regex for the Class also the same Regex is issued. Next, let's complicate the query a bit. Let's combine it with the operators covered above.

Fetch Students with names as student1, student2 and who are male students with age >=25. The command for this is as follows:

```
>db.students.find({"Name":/(student*)/i,"Age":{"$gte":25},"Gender":"M"})
```

{ "_id" : ObjectId("52f89eb1e451bb7a56e59085"), "Name" : "Student1", "Age" : 30, "Gender" : "M", "Class" : "Biology", "Score" : 90 }

{ "_id" : ObjectId("52f89ecae451bb7a56e59086"), "Name" : "Student2", "Age" : 30, "Gender" : "M", "Class" : "Chemistry", "Score" : 90 }

MapReduce :

Question 1: Explain MapReduce framework of MongoDB with example.

MapReduce is a data processing paradigm for condensing large volumes of data into useful aggregated results. In MongoDB, MapReduce is used for processing large data sets in a parallel, distributed manner. It consists of two phases

1. **Map Phase**: In this phase, the map function processes each document and emits key-value pairs.
2. **Reduce Phase**: The reduce function then takes all the values associated with the same key and merges them to produce a single result.

Syntax :

```
db.collection.mapReduce(
  mapFunction,
  reduceFunction,
  {
    out: { merge: "outputCollection" },
    query: { <query condition> },
    sort: { <sort specification> },
    limit: <number>,
    finalize: finalizeFunction,
    scope: { <variables accessible in map/reduce/finalize> },
    jsMode: true|false,
    verbose: true|false
  }
)
```

- **mapFunction**: A JavaScript function that maps the documents to key-value pairs.
- **reduceFunction**: A JavaScript function that reduces the values for a given key to a single result.
- **outputCollection**: The collection where the result will be stored.
- **query**: Optional query to filter the documents.
- **sort**: Optional sorting of documents before processing.
- **limit**: Optional limit on the number of documents.
- **finalizeFunction**: Optional function to perform any final modifications to the result of the reduce function.
- **scope**: Optional object that specifies variables that can be accessed within the map, reduce, and finalize functions.

- **jsMode**: Boolean that enables JavaScript mode for the map/reduce functions.
- **verbose**: Boolean that specifies whether to include detailed information in the result.

Example

Suppose we have a collection `sales` with documents that store information about sales transactions:

```
{ "_id": 1, "item": "apple", "quantity": 10, "price": 2 }
{ "_id": 2, "item": "banana", "quantity": 5, "price": 1 }
{ "_id": 3, "item": "apple", "quantity": 15, "price": 2 }
{ "_id": 4, "item": "banana", "quantity": 7, "price": 1 }
```

We want to calculate the total quantity sold for each item.

```
// Map function
var mapFunction = function() {
  emit(this.item, this.quantity);
};

// Reduce function
var reduceFunction = function(key, values) {
  return Array.sum(values);
};

// Execute MapReduce
db.sales.mapReduce(
  mapFunction,
  reduceFunction,
  {
    out: "totalQuantity"
  }
);

// View the result
db.totalQuantity.find();
```

Example Explanation

1. **Map Function**: The `mapFunction` processes each document and emits the `item` field as the key and the `quantity` field as the value. For example, for the first document, it emits `("apple", 10)`.

2. **Reduce Function**: The `reduceFunction` takes all the values (quantities) for each unique key (item) and sums them. For instance, it sums the quantities for "apple" as `10 + 15 = 25`.

3. **Output**: The result is stored in the `totalQuantity` collection. After running the MapReduce operation, the `totalQuantity` collection will contain documents like:

```
{ "_id": "apple", "value": 25 }
{ "_id": "banana", "value": 12 }
```

This indicates that the total quantity of apples sold is 25, and the total quantity of bananas sold is 12.

Aggregate :

- The `aggregate` method in MongoDB is used for processing data records and returning computed results.
- It performs operations like filtering, grouping, sorting, reshaping, and modifying documents that pass through the pipeline.
- The `aggregate` method is powerful and flexible, allowing for complex data manipulations and analysis within the database.

Syntax :

db.collection.aggregate(pipeline, options)

- **pipeline**: An array of stages, each of which represents an operation to be performed on the documents. Stages are processed sequentially.
- **options**: An optional document that specifies options to the aggregation, such as `allowDiskUse`, `cursor`, `maxTimeMS`, etc.

Example

Suppose we have a `sales` collection with documents that represent sales transactions:

```
{
 "_id": 1,
 "product": "Laptop",
 "amount": 1200,
 "date": "2023-06-01"
}
```

```
 {
  "_id": 2,
  "product": "Phone",
  "amount": 800,
  "date": "2023-06-02"
 }
 {
  "_id": 3,
  "product": "Laptop",
  "amount": 1300,
  "date": "2023-06-03"
 }
 {
  "_id": 4,
  "product": "Tablet",
  "amount": 600,
  "date": "2023-06-04"
 }
```

We want to calculate the total sales amount for each product. Here is how we can do this using the `aggregate` method:

```
db.sales.aggregate([
 {
  $group: {
   _id: "$product",
   totalSales: { $sum: "$amount" }
  }
 }
])
```

Example Explanation

1. **$group Stage**: This stage groups the documents by the `product` field. The `_id` field in the `$group` stage is used to specify the field by which to group the documents. Here, `"_id": "$product"` groups the documents by the `product` field.

2. **totalSales Field**: The `totalSales` field is created to store the total sales amount for each product. The `$sum` operator is used to sum the `amount` field of all documents in each group.

Result:

```
[
```

```
  { "_id": "Laptop", "totalSales": 2500 },
  { "_id": "Phone", "totalSales": 800 },
  { "_id": "Tablet", "totalSales": 600 }
]
```

- The result shows that the total sales for `Laptop` is 2500, for `Phone` is 800, and for `Tablet` is 600.

Designing an Application's Data Model :

In this section, you will look at how to design the data model for an application. The MongoDB database provides two options for designing a data model: the user can either embed related objects within one another, or it can reference each other using ID. In this section, you will explore these options.

Designing a data model in MongoDB involves defining how data will be stored, organized, and accessed within the database. Here are the key steps to design an effective data model:

1. **Understand Application Requirements**
   - Identify what data needs to be stored.
   - Determine how the data will be used and accessed.
   - Example: For an e-commerce application, you might need to store information about users, products, orders, and reviews.
2. **Identify Entities and Relationships**
   - Identify the main entities (collections) and their attributes (fields).
   - Determine relationships between entities.
   - Example: Entities could be users, products, orders, and reviews. Relationships include users writing reviews and placing orders.
3. **Choose the Data Model Type**
   - Decide between embedding (nested documents) and referencing (linked documents) based on access patterns and data size.
   - Example: Embed user address information within the user document if addresses are infrequently changed. Reference products in orders if products are large and frequently accessed.
4. **Define Schema for Collections**
   - Define the fields for each collection, including their data types.
   - Example:
     - Users collection: { _id, name, email, address }
     - Products collection: { _id, name, price, description }
     - Orders collection: { _id, user_id, product_ids, total_amount, order_date }
     - Reviews collection: { _id, user_id, product_id, rating, comment }

5. **Consider Indexing for Performance**
   - Identify fields that require indexing to improve query performance.
   - Example: Index email field in the users collection for fast lookup. Index product_id in the reviews collection for quick access to reviews for a product.

Example Application Data Model

Let's design a simple data model for a blog application with the following requirements:

- Users can write multiple posts.
- Posts can have multiple comments.
- Users can like posts.

Collections and Schemas:

1. **Users Collection**
   json
   Copy code
   ```
   {
     "_id": "user123",
     "name": "John Doe",
     "email": "john@example.com"
   }
   ```

2. **Posts Collection**
   json
   Copy code
   ```
   {
     "_id": "post123",
     "user_id": "user123",
     "title": "My First Post",
     "content": "This is the content of my first post.",
     "likes": ["user456", "user789"]
   }
   ```

3. **Comments Collection**
   ```
   {
     "_id": "comment123",
     "post_id": "post123",
     "user_id": "user456",
     "comment": "Great post!"
   }
   ```

Explanation:

1. **Users Collection:**
   - Stores user information.
   - Fields: _id (unique identifier), name, and email.
2. **Posts Collection:**
   - Stores blog posts.
   - Fields: _id (unique identifier), user_id (reference to the user who wrote the post), title, content, and likes (array of user IDs who liked the post).
3. **Comments Collection:**
   - Stores comments on posts.
   - Fields: _id (unique identifier), post_id (reference to the post), user_id (reference to the user who wrote the comment), and comment.

Example Query and Explanation:

- **Query to Find All Posts by a User:**
  db.posts.find({ user_id: "user123" })

Explanation:

- This query searches the posts collection for all documents where the user_id field is "user123".
- It returns all posts written by the user with ID user123.

Relational Data Modelling and Normalization:

Relational data modeling and normalization in MongoDB involve organizing the database schema to minimize redundancy and ensure data integrity. Although MongoDB is a NoSQL database and follows a flexible schema design, principles of relational data modeling can still be applied for better organization and performance.

Key Points

1. **Data Modeling**:
   - **Embedding**: Storing related data in a single document. This is suitable for one-to-few or one-to-many relationships.
   - **Referencing**: Storing related data in separate documents and linking them using references. This is suitable for one-to-many or many-to-many relationships.
2. **Normalization**:

- **1st Normal Form (1NF)**: Ensure that each field contains only atomic (indivisible) values and each record is unique.
- **2nd Normal Form (2NF)**: Ensure that all non-key attributes are fully functional dependent on the primary key.
- **3rd Normal Form (3NF)**: Ensure that all non-key attributes are not dependent on other non-key attributes.

Example

Suppose we have two collections: students and courses.

1. **Embedding Example**:
- For storing the courses a student is enrolled in within the students document.

```
{
 "_id": 1,
 "name": "John Doe",
 "age": 20,
 "courses": [
  {
   "course_id": 101,
   "course_name": "Mathematics",
   "credits": 4
  },
  {
   "course_id": 102,
   "course_name": "Physics",
   "credits": 3
  }
 ]
}
```

2. **Referencing Example**:
- For storing course details in a separate courses collection and referencing them in the students collection.

students Collection:

```
{
 "_id": 1,
 "name": "John Doe",
 "age": 20,
 "course_ids": [101, 102]
}
```

courses Collection:

```
{
```

```
  "_id": 101,
  "course_name": "Mathematics",
  "credits": 4
}
{
  "_id": 102,
  "course_name": "Physics",
  "credits": 3
}
```

Example Explanation

1. **Embedding**:
   - Embedding the courses within the student's document is efficient for queries that need student information along with their courses. However, this approach can lead to data redundancy if course details are updated frequently across multiple students.
2. **Referencing**:
   - Referencing involves storing only the course IDs in the student's document and maintaining course details in a separate collection. This approach normalizes the data, reduces redundancy, and makes it easier to update course details. However, it requires additional queries to fetch course details when needed.

MongoDB Document Data Model Approach :

MongoDB uses a document-oriented data model, which allows for flexible and scalable database design. This approach is different from traditional relational databases and offers several benefits.

1. Document Structure

- **Point**: Data is stored in BSON (Binary JSON) format, which is a binary representation of JSON-like documents.
- **Explanation**: Each document is a set of key-value pairs, similar to JSON objects, making it easy to map to objects in most programming languages.

2. Schema Flexibility

- **Point**: MongoDB documents can have varying structures.
- **Explanation**: Unlike relational databases, MongoDB does not require a fixed schema, allowing each document to have a different set of

fields. This flexibility makes it easier to evolve the data model over time without requiring extensive database migrations.

3. Embedded Documents and Arrays

- **Point**: MongoDB allows embedding documents and arrays within documents.
- **Explanation**: This feature supports complex data structures and relationships within a single document, reducing the need for joins and improving read performance.

4. Collections

- **Point**: Documents are grouped into collections.
- **Explanation**: A collection is analogous to a table in a relational database but does not enforce a schema. Collections organize documents and allow for efficient querying and indexing.

5. Indexes

- **Point**: MongoDB supports indexes to improve query performance.
- **Explanation**: Indexes can be created on any field within a document, allowing for fast retrieval of data. Compound indexes and unique indexes are also supported.

Example Explanation

Suppose we have a simple application to manage information about books and their authors. Here's how you might structure this data using MongoDB's document data model:

Example Document:

```
{
  "title": "The Great Gatsby",
  "author": {
    "firstName": "F. Scott",
    "lastName": "Fitzgerald"
  },
  "publicationYear": 1925,
  "genres": ["Classic", "Fiction"],
  "ratings": [
    { "user": "Alice", "score": 5 },
    { "user": "Bob", "score": 4 }
  ]
}
```

Explanation of Example:

1. **Document Structure**:
   - The entire book's information is stored in a single document. This includes the title, author, publication year, genres, and ratings.
2. **Schema Flexibility**:
   - If a new book has additional fields, such as an ISBN number or publisher, those can be added without affecting other documents in the collection.
3. **Embedded Documents and Arrays**:
   - The author field is an embedded document, containing the author's first and last names. This allows us to store related information in a single document.
   - The genres field is an array of strings, representing the different genres the book belongs to.
   - The ratings field is an array of embedded documents, each containing a user and their score. This efficiently groups all ratings for the book within the same document.
4. **Collections**:
   - All book documents are stored in a books collection. Each document within this collection can have a different structure, as needed.
5. **Indexes**:
   - An index could be created on the title field to quickly search for books by title.
   - Another index could be created on the author.lastName field to efficiently search for books by the author's last name.