Develop a Reliable Backend with Node and Express

Operations in MongoDB

A Day in the Life of a MERN Stack Developer

John is a MERN stack developer tasked with working on a trade and commercial services project. The project handles vast trade and commercial services data in a sizable database.

The end users require quick access to the data without loading large files into memory. To meet these requirements, he uses CRUD operations.

This lesson will enable him to comprehend all the essential operations required to fulfill the requirements. CRUD operations are used to manage large files with minimal memory usage.

Learning Objectives

By the end of this lesson, you will be able to:

- Analyze the **Create** operation in MongoDB to comprehend how it impacts database performance and resource utilization
- Describe the concepts of the Read operation in MongoDB for facilitating efficient data querying and retrieval strategies
- Demonstrate how to update fields in documents using various methods providing insights into the approaches available for modifying data
- Assess the operations in MongoDB to gauge the effectiveness of MongoDB in handling increasing data loads

Create Operation

Create Operation

The **Create** operation creates or adds new data structures to the database.

It allows users to create collections, documents, indexes, views, and other structures that can be used to store and manipulate data.

Creating Documents: Overview

Creating documents is an essential part of working with MongoDB as it allows the user to add data to collections and build up their database structure.

Documents are stored in collections, where each document consists of one or more fields.

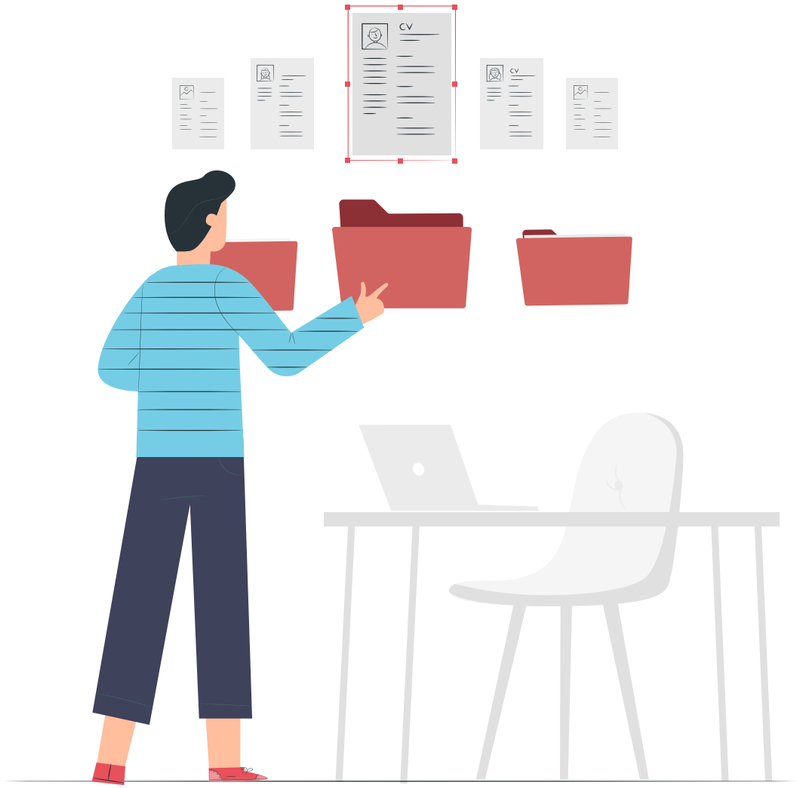
Ordered Inserts

The ordered inserts refer to the ability to insert multiple documents into a collection while maintaining the order of the inserted documents.

This means that the documents will be inserted into the collection in the same order in which they were specified in the insert command.

Write Concern

The write concern pertains to the level of acknowledgment that a client requests when executing write operations.



Write Concern

The following are the possible values for the write concern parameter:

1

{ w: 0 }: This option is the fastest but least durable.

2

{ w: 1 }: This option is the default write concern in MongoDB.

3

{ w: "majority" }: This option is more durable than w:1 and is recommended for most production environments.

Write Concern

The following are the possible values for the write concern parameter:

4

{ w: <number> }: This option allows for customizing the required durability based on the application's specific needs.

5

{ wtimeout: <milliseconds> }: This option sets a timeout for the write operation.

6

{ journal: true }: This option provides additional durability, which can impact performance.

What Is Atomicity?

Atomicity is a characteristic that guarantees that a transaction is treated as an indivisible and single unit of work.

It implies that either all the transaction modifications are fully applied or not.

Importance of Atomicity

Atomicity is significant due to the following reasons:

- To ensure data consistency
- To maintain integrity, especially in multi-user environments

Create and Insert

Problem Statement: **Duration: 20 min.**

You have been assigned a task to create and insert the document on a remote desktop.

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

1. Connect to MongoDB Compass by using virtual machine extensions
2. Create a database in MongoDB Compass
3. Create collections in the database
4. Insert documents in the collection

Read Operation

What Is a Read Operation?

In MongoDB, the read operation retrieves data from the database.

The following methods are used to query a collection and retrieve documents:

01

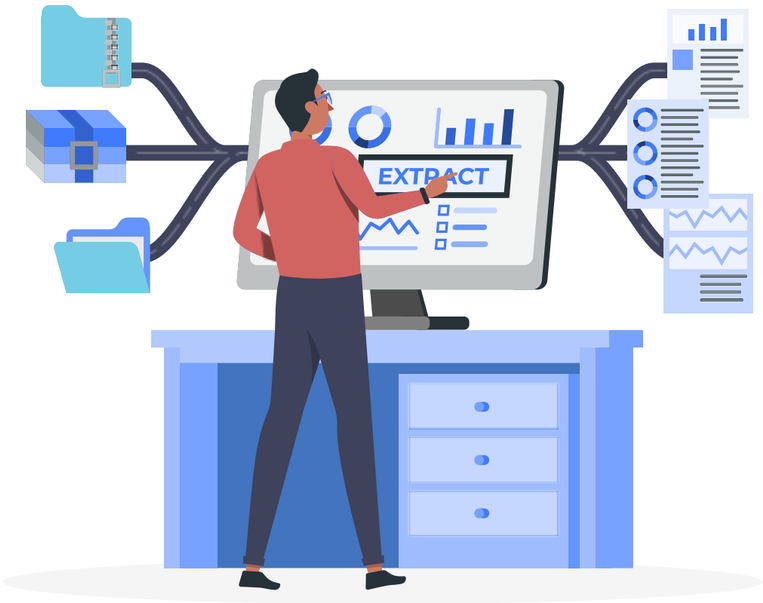
db.collection.find()

02

db.collection.findOne()

db.collection.find()

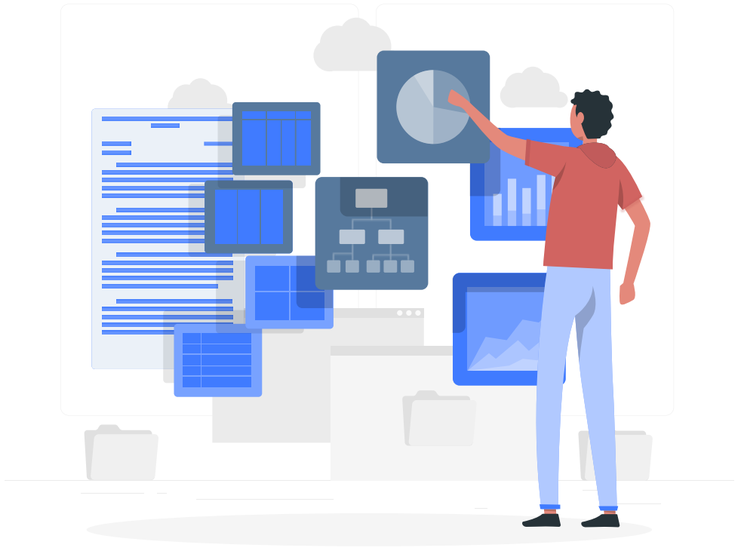
db.collection.find() is a method in MongoDB that retrieves documents from a specified collection.



It takes an optional query object as a parameter, which filters the documents that match specific criteria.

db.collection.findOne()

db.collection.findOne() is a method in MongoDB that retrieves a single document from a specified collection that matches specific query criteria.



The **findOne()** method takes an optional query object as a parameter, which filters the documents that match specific criteria.

Uses of db.collection.findOne()

Some of the uses of findOne() are to:

01

Retrieve a single document from the collection

02

Check whether a document with a specific property or set of properties exists in the collection

03

Retrieve a specific document with a specific value of a property

04

Collect a sample document to get an idea from a collection

Read Operation

Problem Statement: **Duration: 10 min.**

You have been assigned a task to gain basic understanding of JSON and BSON structure for application in MongoDB.

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

1. Connect to MongoDB Compass by using virtual machine extensions
2. Create databases in MongoDB Compass
3. Create a collection in the database
4. Perform Read operation

Update Operation

Update Operation

The **Update** operation modifies the contents of one or more documents in a collection.

The update operators enable users to input various updates on their data.

Update Fields in Documents

There are various methods used to update fields in documents that allow a user to modify one or multiple documents in a collection, depending on requirements.

The three commonly used methods are:

1. updateOne()
2. updateMany()
3. replaceOne()

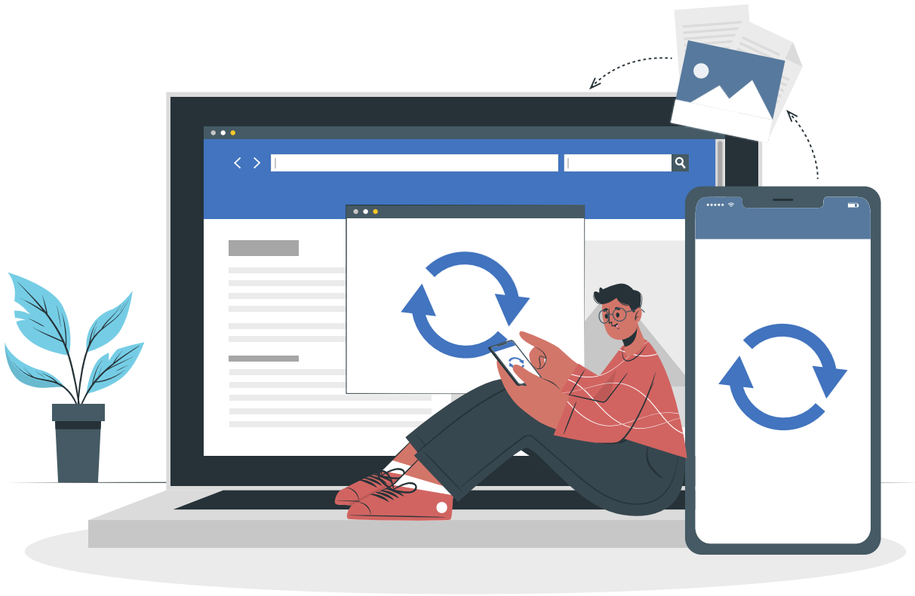
updateOne()

The **updateOne()** method updates a single document at a time that matches specified criteria.

If multiple documents match the filter, only the first will be updated.

updateMany()

The **updateMany()** method updates multiple documents at a time, matching a specified filter.



Syntax:

```
db.collection.updateMany(filter, update, options)
```

replaceOne()

The **replaceOne()** method replaces an old document with a new document that matches a specified filter.

Syntax:

```
db.collection.replaceOne(filter, replacement, options)
```

Update Multiple Fields with $set

The **$set** operator updates one or more fields in a document.

It is possible to modify the value of an existing field or add a new field to the document with this method.

Syntax

The basic syntax to update multiple fields with $set in MongoDB is as follows:

```
db.collection.updateOne(
   { <filter> },
   {
      $set: {
         <field1>: <value1>,
         <field2>: <value2>,
         ...
      }
   }
)
```

Incrementing and Decrementing Values

The **$inc** operator is employed to update numerical values, such as counters in documents, by incrementing or decrementing them.

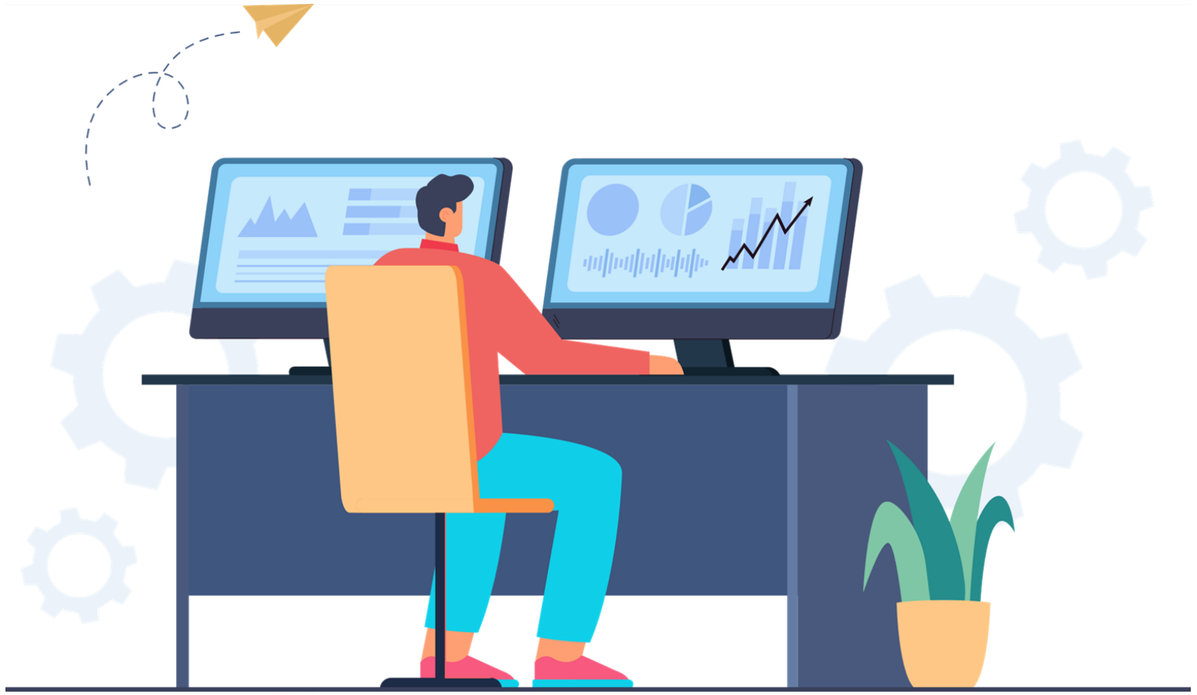
Syntax

The basic syntax for incrementing or decrementing a field using **$inc** in MongoDB is as follows:

```
db.collection.updateOne(
   { <filter> },
   {
      $inc: {
         <field>: <amount>
      }
   }
)
```

Use of $min, $max, and $mul

MongoDB also provides other arithmetic operators besides the **$set** and **$inc** operators, such as **$min**, **$max**, and **$mul**, to update the values of fields in a document.



$min Operator

The **$min** operator updates the field value to a new value; the specified value must be less than the current value.

```
db.collection.updateOne(
   { <filter> },
   {
      $min: {
         <field>: <value>
      }
   }
)
```

If the new value is greater than or equal to the current value, no changes are made to the field.

$max Operator

The **$max** operator updates the field value to a new value; the specified value must be greater than the current value.

```
db.collection.updateOne(
   { <filter> },
   {
      $max: {
         <field>: <value>
      }
   }
)
```

If the new value is less than or equal to the current value, no changes are made to the field.

$mul Operator

The **$mul** operator gives the product of the field value and the specified value.

```
db.collection.updateO
ne(
   { <filter> },
   {
      $mul: {
         <field>:
<value>
      }
   }
)
```

Rename Fields

The rename field is used to rename a field in a document using the **$rename** operator in an update operation.

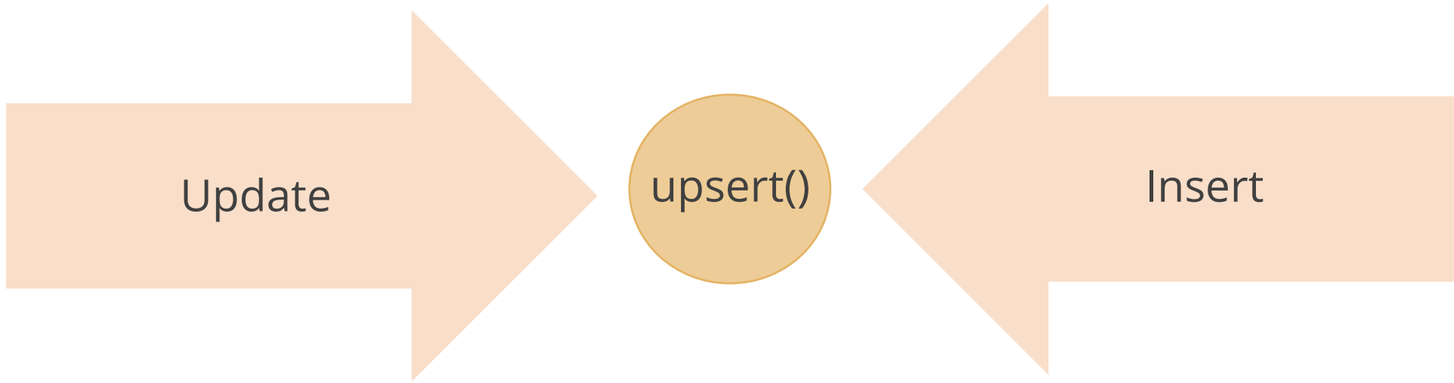
Syntax

The basic syntax for using the **$rename** operator is as follows:

```
db.collection.updateOne(
   { <filter> },
   {
      $rename: {
         <oldFieldName>:
<newFieldName>
      }
   }
)
```

upsert()

The **upsert()** method simplifies the logic of updating and inserting documents.



It is advantageous as it streamlines the process of updating or inserting documents.

upsert(): Syntax

To use the **upsert()** method, a user can specify two arguments:

<query>

<update>

The syntax for **upsert()** is shown below:

```
db.collection.update( <query>, <update>, { upsert: true })
```

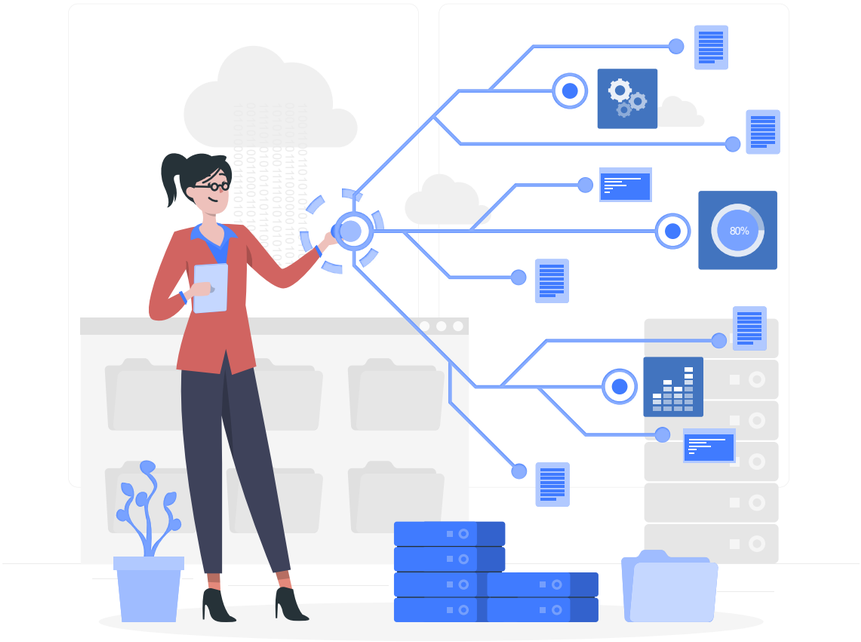
upsert(): Example

Following is an example for **upsert()**:

```
{“_id”: 1, “name”: “John”, “age”: 30 }
{“_id”: 2, “name”: “Mary”, “age”: 25 }

db.users.update( {“_id”: 3},
                {“$set”: {“name”: “Bob”, “age”: 40}},
             {upsert : true}
                )
```

Matched Array Element

The **Matched Array Element** feature updates or removes elements in an array that satisfy a specified condition.



It is done using the **$elemMatch** operator in combination with update or remove operations.

$elemMatch Operator: Syntax

The **$elemMatch** operator selects documents containing an array field with at least one element matching the specified criteria.

The syntax for **$elemMatch** is shown below:

```
db.collection.find({field: { $elemMatch: { criteria } } })
```

$elemMatch Operator: Example

Following is an example to illustrate the usage of the **$elemMatch** operator:

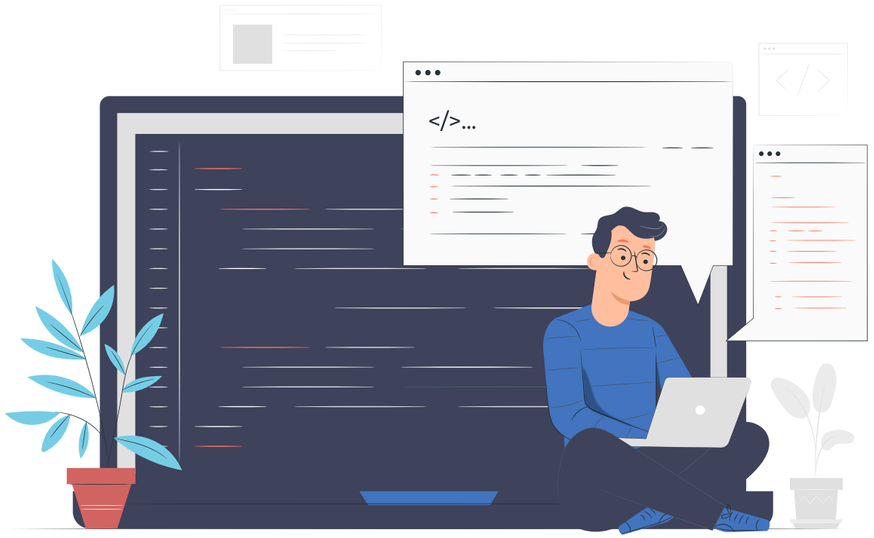
```
#{ "_id": 1,"order_items": [{ "product_name": "iphone
#13","quantity":1,"price": 999},

#{"product_name": "Macbook Pro","quantity":1,"price":
1999}]},

#{"_id": 2,"order_items": [{ "product_name": "Apple
Watch #Seriese","quantity":2,"price": 399}]}

db.orders.finds({ order_items: {$elemMatch: { quality:
{$gte: 2}}}})
```

Result:


```
{
   "_id" : 2,
   "order_items" : [
      {
         "product_name" : "Apple Watch Series 7",
         "quantity" : 2,
         "price" : 399
      }
   ]
}
```

Update All the Array Elements

The **$[ ]** operator and the array update operator are used to update all array elements.



Note:

The update operation will affect all elements in the array that match the query.

Update All the Array Elements: Syntax

The basic syntax to update all arrays is shown below:

```
db.collection.update( { <query> }, { $set: { "array.$[]" : <new
value> } },  { multi: true })
```

Update All the Array Elements: Example

Following is an example to update all the elements in a MongoDB database:

```
#document in collection

# {“_id”: 1, “name”: “John”, “age”: 30, “hobbies”: [“reading”, “running”, “swimming”]}

db.users.update( {_id: 1}, {$set: {“hobbies.$[]” : “hiking” }},
{multi: true})
```

Finding and Updating Specific Fields

If the user wants to find and update specific fields within a document, use the **$set** operator and the **$** positional operator.

The **$** positional operator updates the value of a specific field within an array that matches a given condition.

$set, $push, $pull, and $addToSet: Example

Following is an example of the **$set**, **$push**, **$pull**, and **$addToSet** operators:

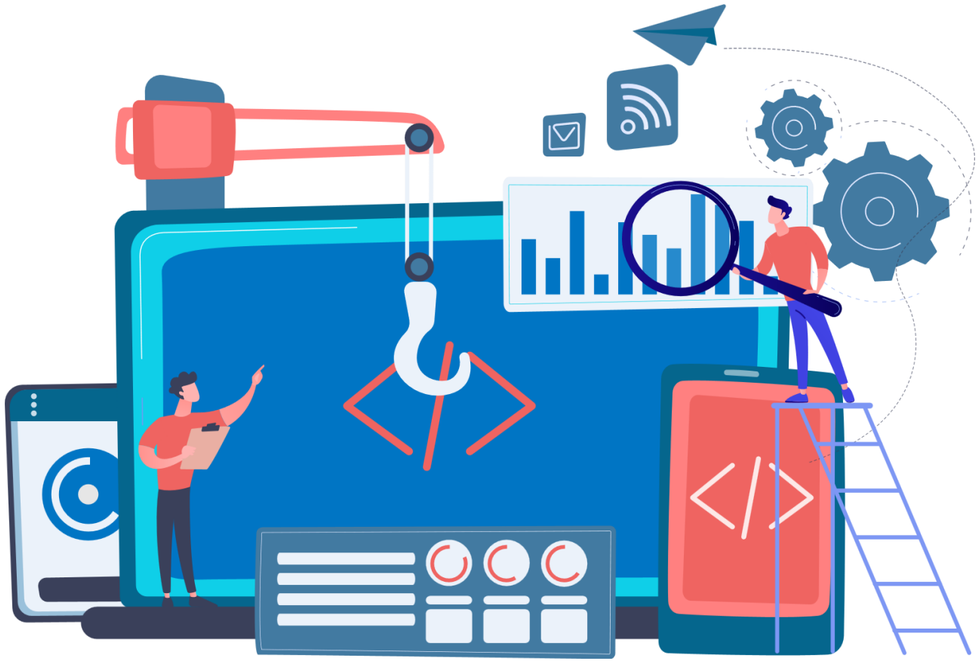
```
#document in the collection
# { "_id": 1, "name": "John", "age": 30, "hobbies":
        [{"name": "reading", "type": "indoor"},
         {"name": "Running", "type": "outdoor"},
         [{"name": "swimming", "type": "outdoor"}]}

#set, push, pull and addtoset command
db.users.update( {_id: 1},{$set:{"hobbies.$[elem1].type": "indoor"},
        $push:{hobbies: {"name":"hiking", "type": "outdoor" }},
        $pull:{hobbies: {"name": "reading"}},
        $addtoSet:{"hobbies.name": "swimming"}},
{arrayFilters: [{"elem1.name": "running"}]})
```

Output:


```
{
  "_id": 1,
  "name": "John",
  "age": 30,
  "hobbies": [
    { "name": "reading", "type": "indoor" },
    { "name": "running", "type": "indoor" },
    { "name": "swimming", "type": "outdoor" },
    { "name": "hiking", "type": "outdoor" }
  ]
}
```

Adding Elements to Array

The **$push**, **$addToSet**, and **$each** operators add elements to an array.



They allow easy modification of documents with complex nested structures.

Syntax

$push

```
db.collection.updateOne({ <query> }, { $push: { <field>: <value> } })
```

$addToSet

```
db.collection.updateOne({ <query> }, { $addToSet: { <field>: <value> } } )
```

$each

```
db.collection.updateOne( { <query> }, { $push: { <field>: { $each: [<value1>, <value2>, ...] } } } )
```

Example

Following is an example of the **$push**, **$addToSet**, and **$each** operators:

```
# sample document
# {“_id”: 1, “name”: “John”, “age”: 30, “hobbies”: [“reading”,
#“running”] }


db.user.updateOne( { _id:1 }, { $push: {hobbies: { $each:
                [“running”, “swimming”, “hiking”]}},
                $addToSet: {hobbies: { $each: [$each:
                [“running”, “swimming”, “hiking” ]}}})
```

Output:

```
["reading", "running", "swimming", "hiking"]
```

Removing Elements from Arrays

The **$pull** and **$pullAll** operators remove elements from an array within a document.

They allow easy modification of documents with complex nested structures.

Syntax

$pull

{ $pull: { <arrayField>: { <queryCondition> } } }

$pullAll

{ $pullAll: { <arrayField>: [<value1>, <value2>, ...] } }

Example

Following is an example of the **$pull**, and **$pullAll** operators:

```
# sample document
# {_id:1, name: “apple”, colors: [“red“, “green”, “yellow”]}
# {_id:2, name: “banana”, colors: [“yellow”]}
# {_id:3, name: “grape”, colors: [“purple”, “green”]}


#command to remove single document
   db.fruits.updateOne({name: “apple”}, {$pull: {colors: “green”}})


# command to remove multiple document
   db.fruits.updateOne ({name: “apple”}, {$pullAll: {colors: [“red”, “green”]}})
```

$addToSet Operator

The **$addToSet** operator adds an element to an array field only if it is not present in the array.

```
# command to use $addToSet operator


db.collection.updateOne ({_id: objectId(“6099f02972d841c4e7404a14”)},
                  { $addToSet: {“student.courses”: “Math”}})
```

Update Operation

Problem Statement: **Duration: 10 min.**

You have been assigned a task to understand the MongoDB Compass GUI to update one or more documents in a collection.

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

1. Connect to the MongoDB Server
2. Update the fields using the edit document function
3. Update multiple documents using the filter option

Delete Operation

deleteOne() and deleteMany() Methods

The **deleteOne()** and **deleteMany()** methods can erase documents from a collection based on specified criteria.

- **DeleteOne():** Removes one document from the collection based on specified filter
- **DeleteMany():** Removes multiple documents from the collection based on the specified filter

deleteOne():

```
db.collection.deleteOne(filter, options)
```

deleteMany():

```
db.collection.deleteMany(filter, options)
```

Delete All Entries in a Collection

To delete all entries in a collection in MongoDB, use the **deleteMany()** method with an empty filter object:

Syntax:

```
db.collection.deleteMany({})
```

Delete All Entries in a Collection

Before deleting all entries in a collection, it is recommended to:

Back up the data and ensure that the deletion is necessary and intentional

Instead of deleting all the entries at once, drop the entire collection and recreate it

Delete Operation in MongoDB

Problem Statement: **Duration: 10 min.**

You have been assigned a task to remove one or more documents from a collection in a MongoDB database.

ASSISTED PRACTICE

Assisted Practice: Guidelines

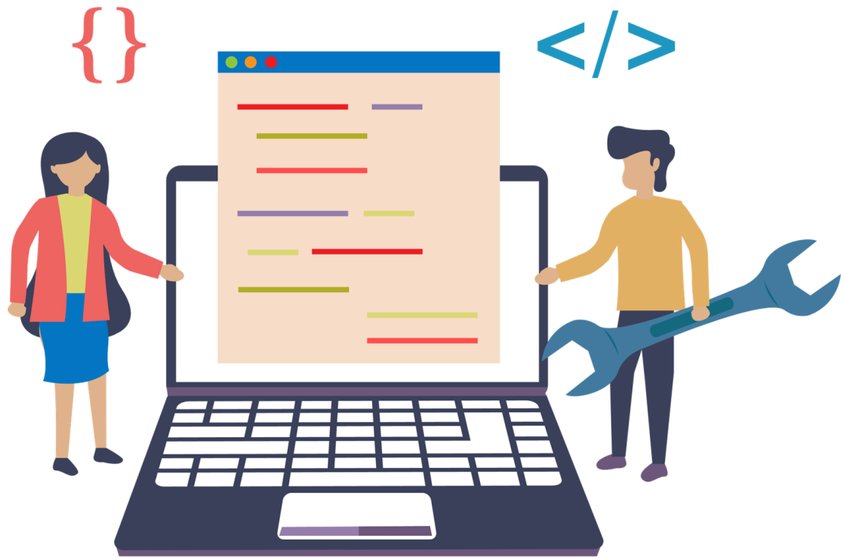
Steps to be followed:

1. Connect to the MongoDB database server
2. Perform the Delete operation

Supporting Operations in MongoDB

Ordered Bulk Insert

This operation is a way to efficiently insert multiple documents into a collection at once, in a specific order.



It inserts many documents with a single write operation.

Example of db.collection.insertMany()

The use of **db.collection.insertMany()** method with an optional ordered parameter set to true:

Code:

```
# command to insert multiple documents in a collection

db.collection.insertMany(

          [
        {_id:1, name: “John”},
        {_id:2, name: “Jane”},
        {_id:3, name: “Jim”}
      ],
      {ordered: true} );
```

Advantages of db.collection.insertMany()

Following are the advantages of **db.collection.insertMany()** method:

1

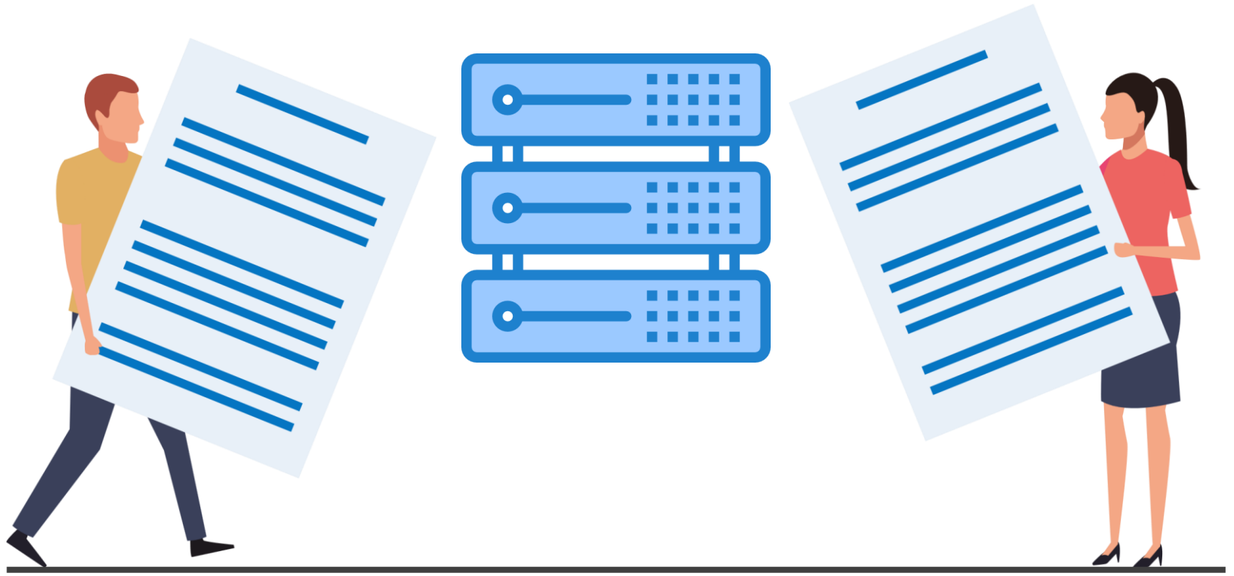
It helps in scenarios where one needs to insert large amounts of data.

2

It is useful if a user wants to ensure that the data is inserted in a consistent and reliable manner.

Unordered Bulk Insert

It allows a user to insert multiple documents into a collection at once without specifying an order.



It inserts all the documents in parallel, using multiple threads or processes.

Example of an Unordered Bulk Insert

Use the **insert_many()** method to perform an unordered bulk insert in MongoDB:

Code:

```
# from pymongo import MongoClient

client = MongoCLient()

db = client ['mydatabase']
collection = db ['mycollection']

documents = [ {'name': 'Alice', 'age': 25}, {'name': 'Bob', 'age': 30} ]

Result = collection.insert_many(docuemnts, ordered = False)
```

Advantages of insert_many()

Following are the advantages of **insert_many()** method:

1

It is useful to improve user application's performance, scalability, and maintainability, mainly when dealing with large datasets.

2

It reduces latency of the insert operation and simplifies the code. Even if the documents are inserted parallelly, an unordered bulk insert operation is still atomic.

Retrieve Document

Retrieve document fetches documents from a collection in a MongoDB database.

Syntax:

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

Example for Document Retrieval

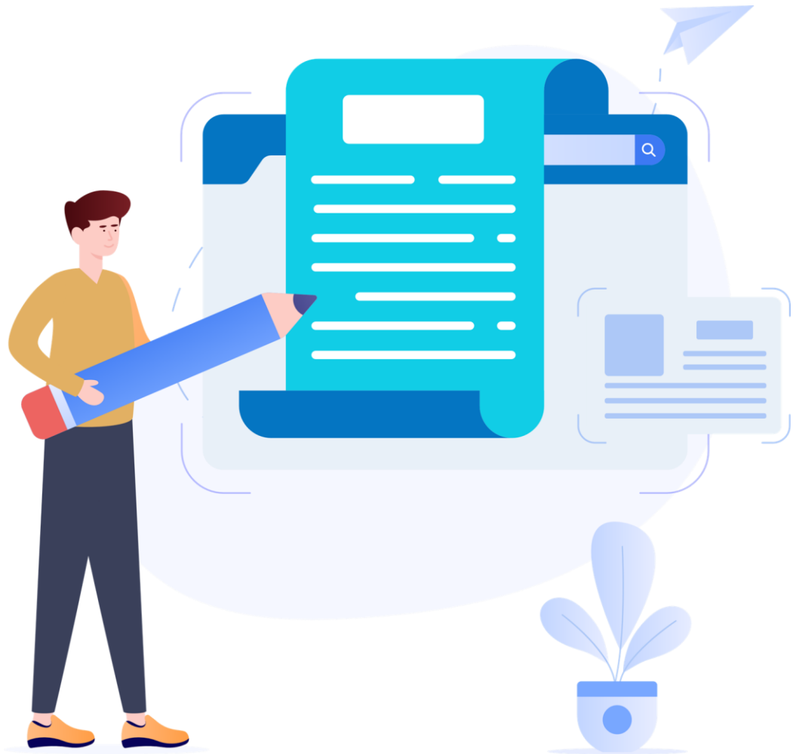
It performs the document retrieval operation using the **db.users.find()** method:

Code:

```
#Example to retrieve the document

db.user.find()

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

db.users.find({}, {password:0})
```

Update Operation

The **update** operation modifies existing documents in a collection.



Update Operation

There are two methods to modify a document:

01

updateOne()

02

updateMany()

Update Operation: Syntax

Following is the syntax of the methods to modify a document:

- Use the **updateOne()** method to update a single document in a collection:

```
db.collection.updateone(filter, update, option)
```

- Use the **updateMany()** method to update multiple documents:

```
db.collection.updateMany(<filter>, <update>)
```

Update Operation: Example

Following is the example of the methods to modify a document:

- Use **db.users.updateOne()** to update a single document:

```
db.users.updateOne({name:”John”}, {$set {age: 30}})
```

- Use **db.users.updateMany()** to update multiple documents:

```
db.users.updateMany({age: {$lt: 18}}, { $inc: { age:1 }})
```

Replace the Document

The replace document operation in MongoDB replaces an existing document in a collection with a new document.

Example of Replace Document

Following is the syntax of the methods to replace a document:

- Use the **db.collection.replaceOne()** method to replace a single document in a collection:

```
db.collection.replaceOne(filter, replacement, options)
```

- Use the **db.collection.replaceMany()** method to replace multiple documents in a collection:

```
db.collection.replaceMany(filter, replacement, options)
```

Example of Replace Document

Following is the example of the methods to replace a document:

- Use **db.collection.replaceOne()** method to replace a single document:

```
db.collection.replaceOne ({name: “John”}, {name: “John Doe”,
age: 30});
```

- Use **db.collection.replaceMany()** method to replace multiple documents:

```
db.collection.replaceMany ({age: {$lt: 18}}, {isMinor: true});
```

Remove the Document

The remove operation in MongoDB deletes one or more documents from a collection.

It is similar to the SQL DELETE command used in relational databases.

Remove the Document

There are two methods available for the remove operation:

deleteOne()

```
# Syntax for deleteOne
db.collection.deleteOne(<filter>, <options>)
```

deleteMany()

```
# syntax for deleteMany
db.collection.deleteMany(<filter>, <options>)
```

Example of Remove Document

- Use the **db.collection.deleteOne()** method to replace a single document:

```
db.employee.deleteOne({employee_id: “1001”})
```

- Use the **db.collection.deleteMany()** method to replace multiple documents:

```
db.employee.deleteMany({ grade: “D” })
```

Operation in MongoDB

Problem Statement: **Duration: 30 min.**

You have been assigned a task to perform operations like inserting, retrieving, updating, and deleting documents in a MongoDB database using Compass.

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

1. Perform bulk Insert operation
2. Retrieve the documents
3. Update the documents
4. Replace the documents
5. Remove the documents

Key Takeaways

- The many-to-one relationships can be implemented using the reference data model.
- To update a specific element in an array, positional operator **$** is used.
- Ordered and unordered bulk inserts efficiently insert the documents into MongoDB collections.
- The replace documents operation allows users to replace an existing document in a collection completely.

Thank You