**DataBase**

The process of storing data in any container is called database

**Types of Database Management System**

1. **Relational Database Management System (RDBMS)/ SQL Database**
2. **Non-Relational Database Management System (NRDBMS) / NoSQL Database**

**Relational Database Management System/ SQL Database**

Relational database means the process of storing the data in format of structured way (in format of organized way).

In RDBMS Data will be stored in a form of tables, row and columns

RDBMS will be managed with different containers / servers like MYSQL, ORACLE, POSTGRES SQL, Heidi SQL etc.,

**Non-Relational Database Management System/ NoSQL Database**Non-Relational Database means the process of storing the data in unorganised way/ unstructured way.

**NRDBMS** will be managed in cloud servers like Mongo cluster, AWS, Azure SQL, Google cloud etc.

NRDBMS are mongo dB, Azure sql, Cassandra DB etc.

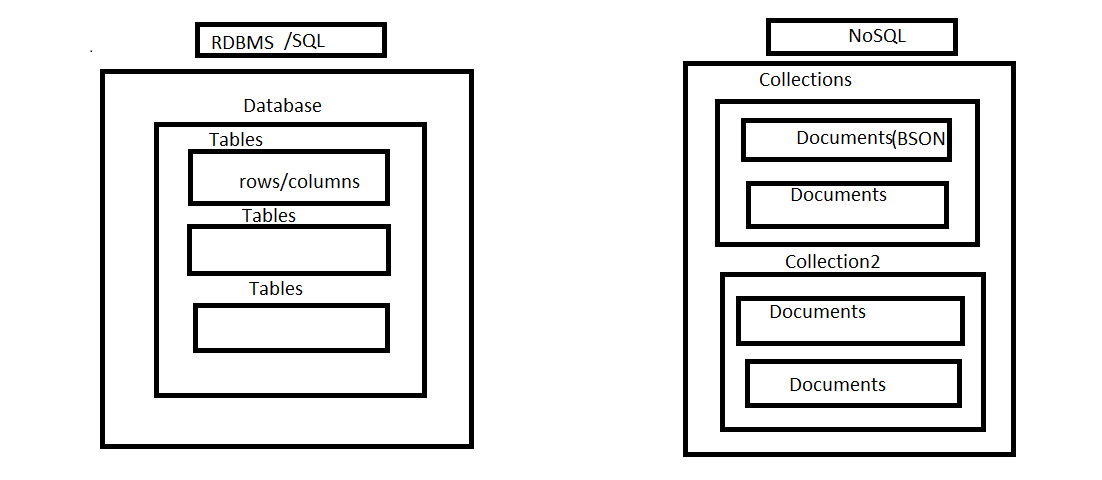
**MongoDB**

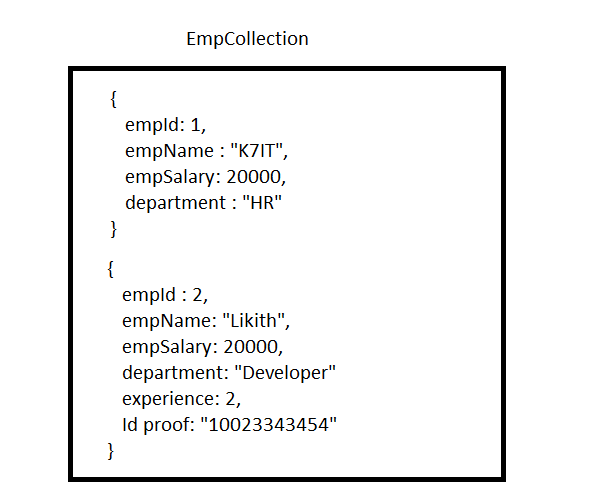
**MongoDB Introduction**

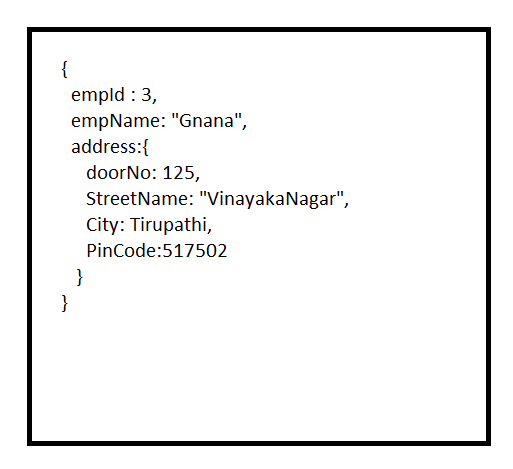
MongoDB is one of the **most important NoSQL databases.**

MongoDB is a document database. It stores data in a type of JSON format called BSON.

A record in MongoDB is a document, which is a data structure composed of field and value pairs. MongoDB documents are similar to JSON objects.







Nested Document

|  |
| --- |
| {  title: "Post Title 1",  body: "Body of post.",  category: "News",  likes: 1,  tags: ["news", "events"], Arrays  date: Date()  } |

|  |
| --- |
| {  "id": 1,  " Name ": "K7IT",  "Salary ": 25000,  " Department": "Development",  "Technologies": [  "java",  "DB",  "HTML",  "CSS",  "JavaScript"  ],  "EducationDetails": {  "ssc": 2008,  "12th": 2010,  "Graduation": 2014  }  } |

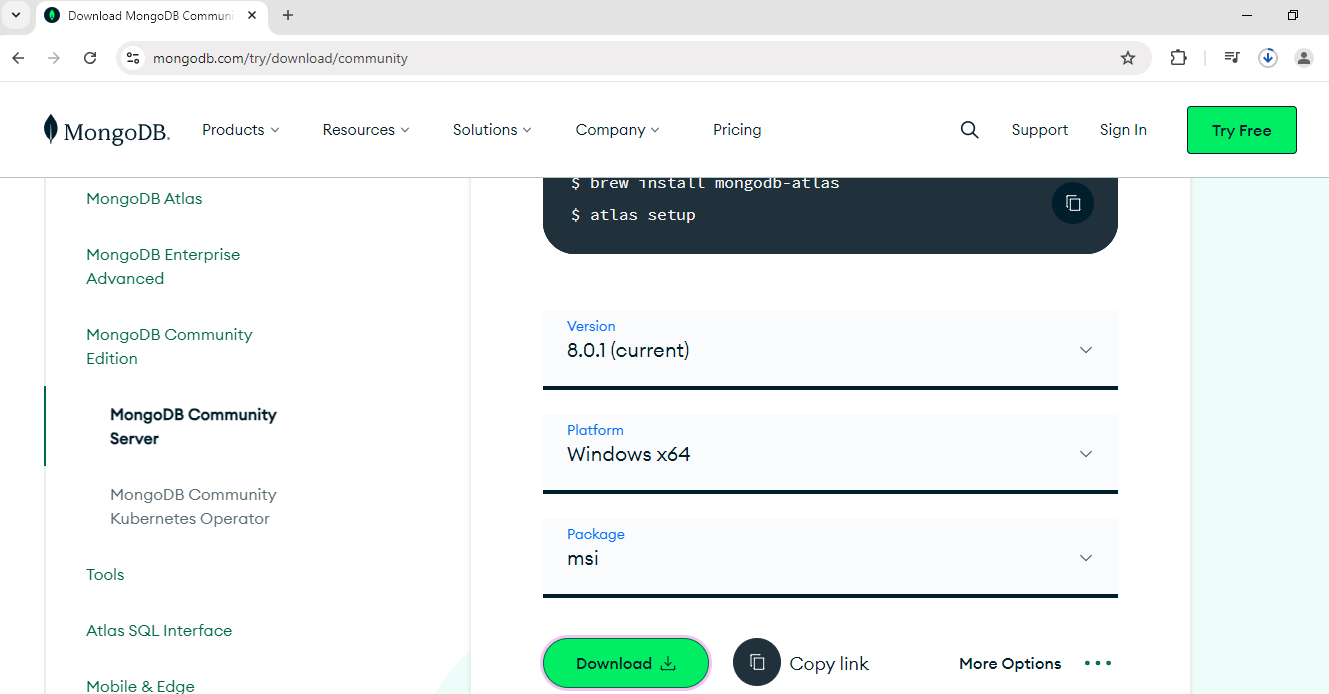
Records in a MongoDB database are called documents, and the field values may include numbers, strings, booleans, arrays, or even nested documents.

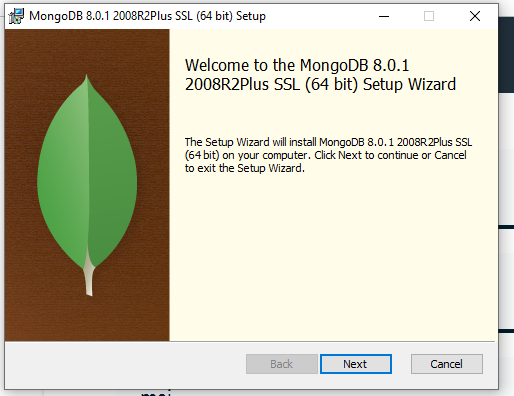
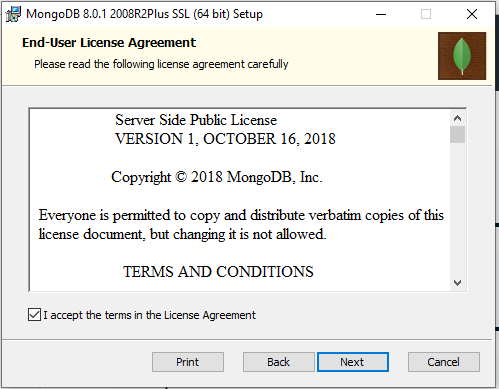
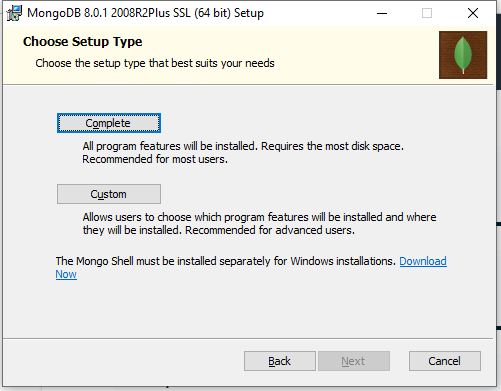
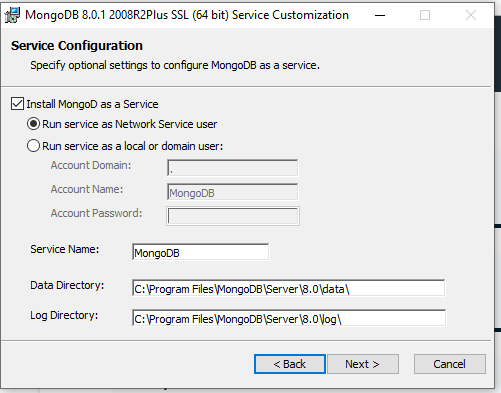
**The advantages of using documents are:**

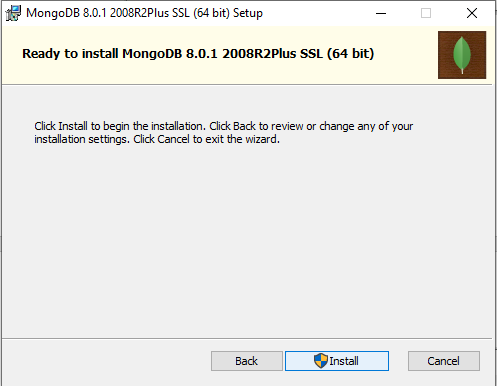
1. It is Open source tool.
2. It is more flexible to store the data.
3. It can store large size of data.
4. Directly we can add the additional data to an existing documents.

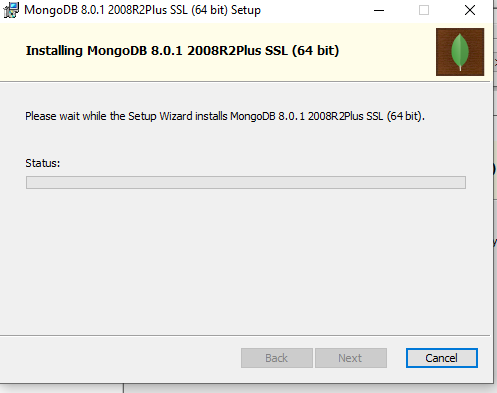
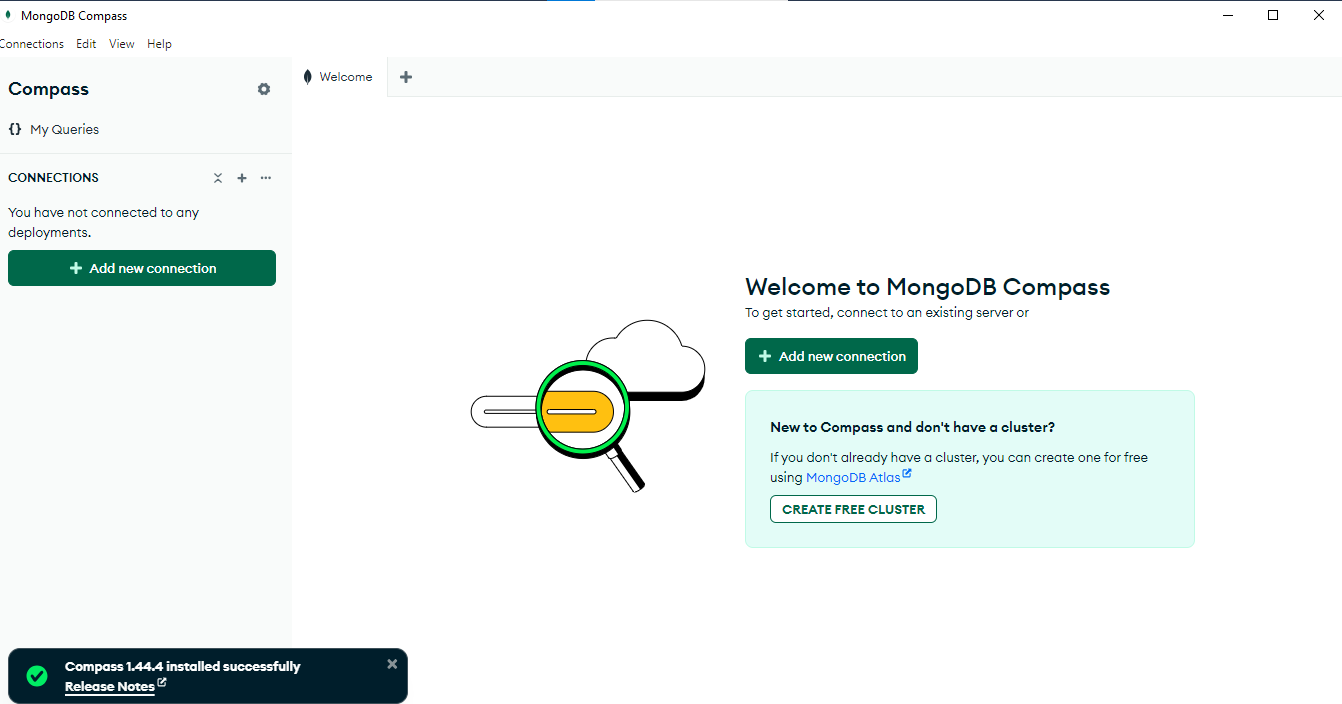
**Installation of MongoDB**

* Go to <https://www.mongodb.com/try/download/community>

****

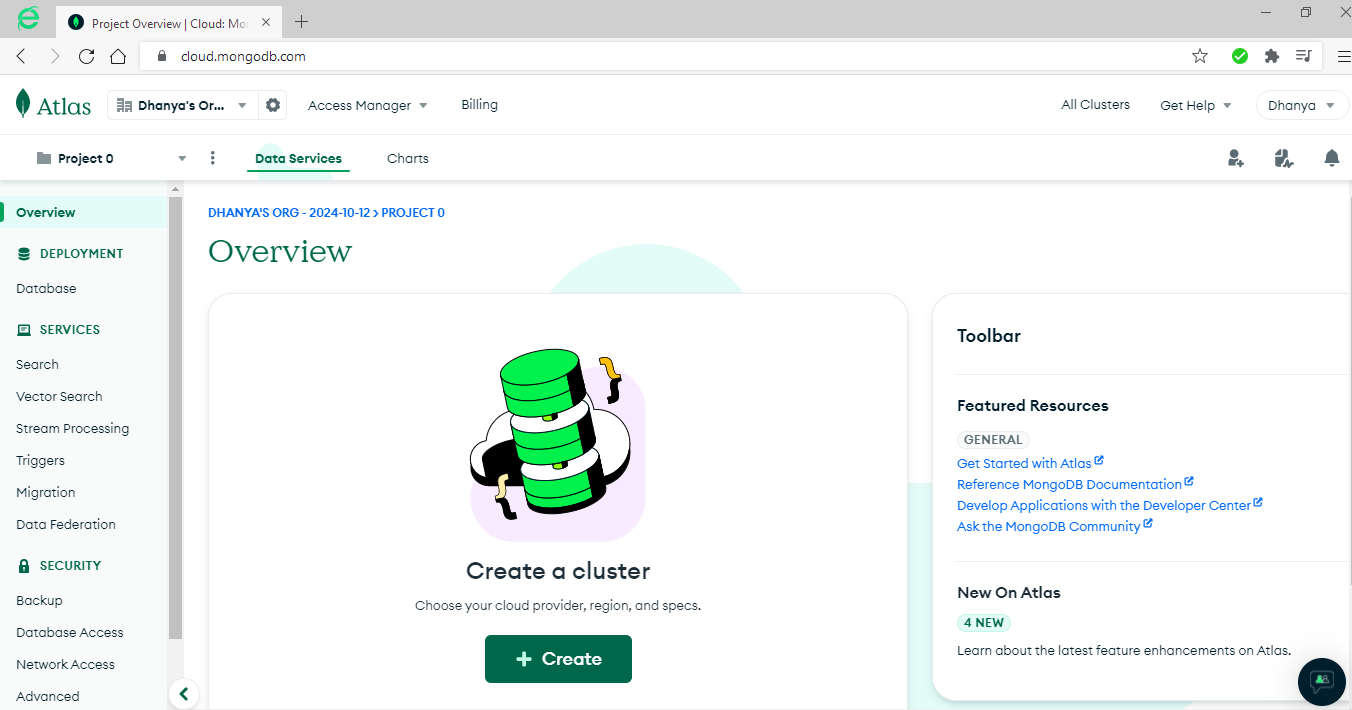
* **Click on Download option**
* ****
* ****
* ****
* Choose complete
* ****
* Choose the run service as Network Service User

****

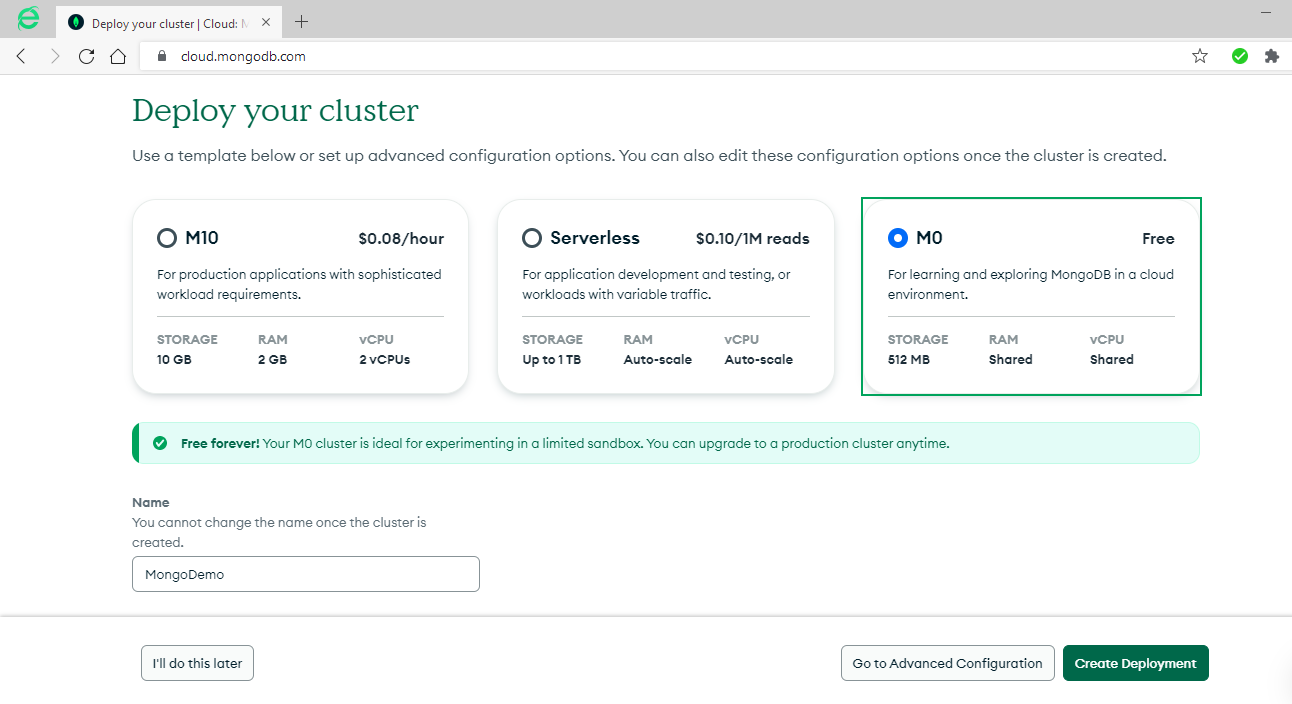
* **Go with Install   
  **
* Once Installation is completed MongoDB compass will open. MongoDB Compass is an editor to connection and write the queries.
* 

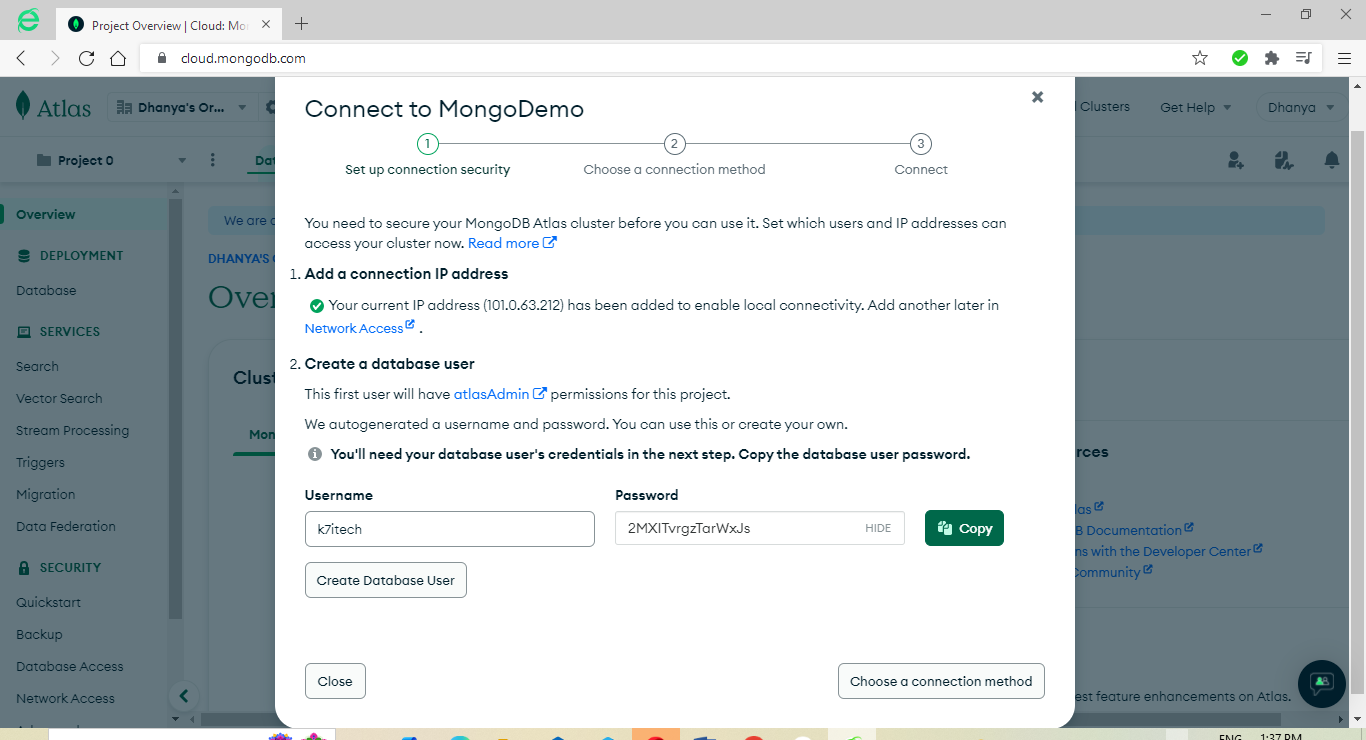
**Create Free Cluster**

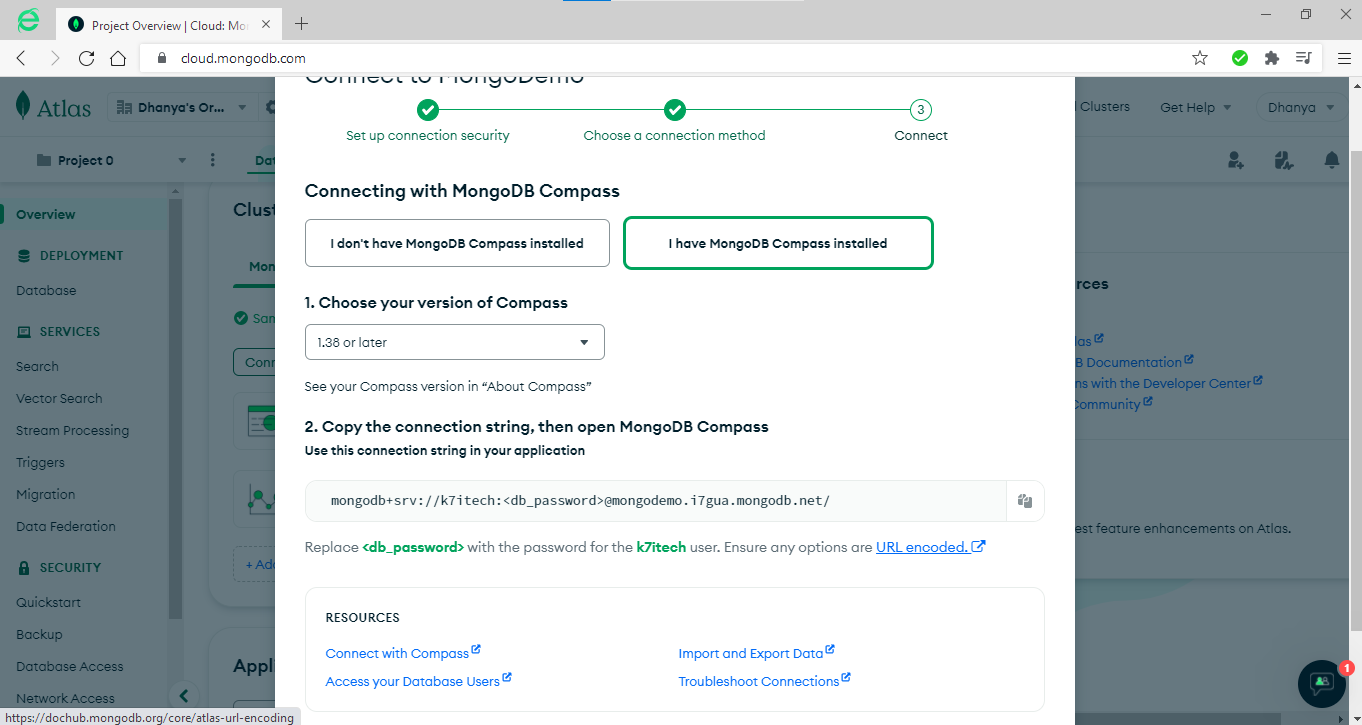
1. Click on Create Free cluster
2. It will navigate to Mongo Db Atlas cloud page. Click on Start free
3. Sign up / register the account with mail Ids.
4. After Signup, verify the mail and login again.

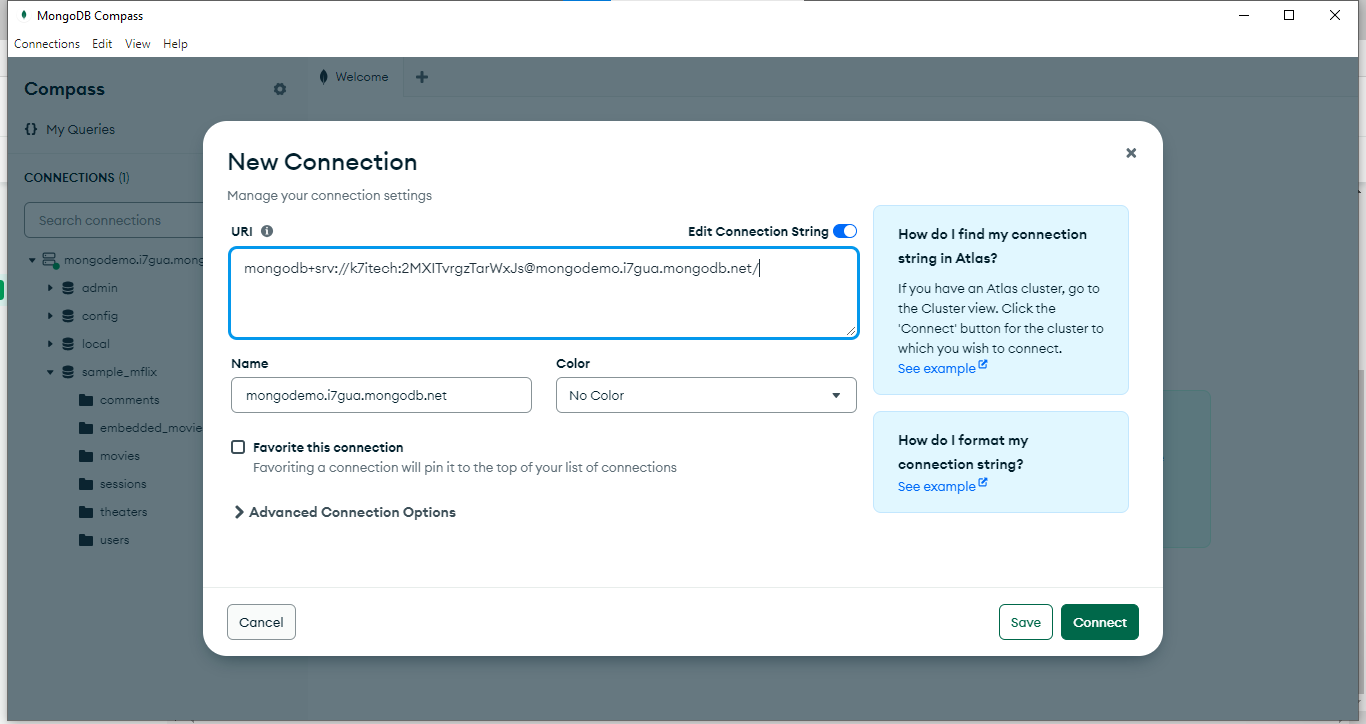
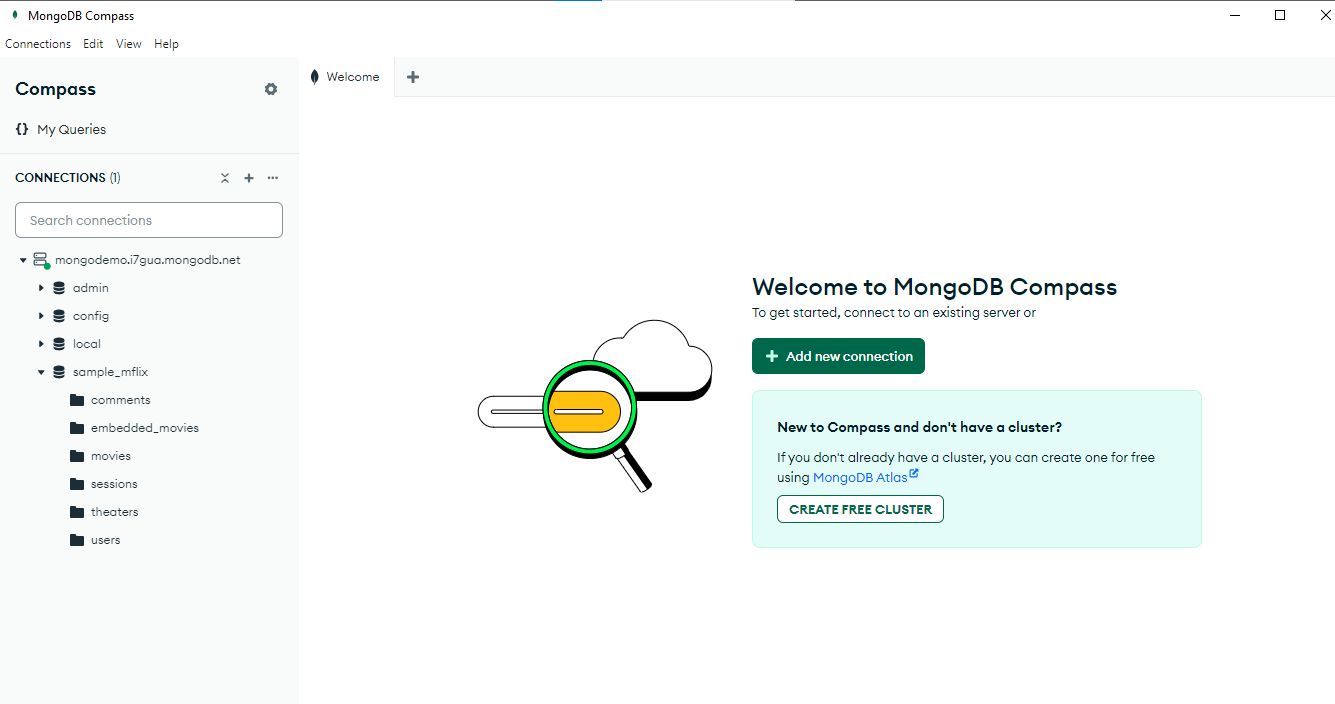


1. Click on create Cluster



1. Choose free and rename the cluster name , choose the server AWS/ GoogleCloud/ Azure and choose the Region -> click on Create Deployment
2. 



1. GO to MongoDB compass and Add Connection and give your connection-string and connect.
2. 
3. Connect to the cloud server.
4. 

**Create Database**  
To create database in mongodb   
 🡪 use databasename

Use flights

To check your currently selected database, use the command **db**

* Db

flights

If you want to check your databases list, use the command **show dbs**

* Show db
* K7ITDemoDB 8.00 KiB
* sample\_mflix 112.62 MiB
* admin 296.00 KiB
* local 31.98 GiB

The dropDatabase() Method

* MongoDB **db.dropDatabase()** command is used to drop a existing database.
* db.dropDatabase()

{ ok: 1, dropped: 'flights' }

## The createCollection() Method

MongoDB **db.createCollection(name, options)** is used to create collection.

### **Syntax**

Basic syntax of **createCollection()** command is as follows −

db.createCollection(name, options)

In the command, **name** is name of collection to be created. **Options** is a document and is used to specify configuration of collection.

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Type** | **Description** |
| Name | String | Name of the collection to be created |
| Options | Document | (Optional) Specify options about memory size and indexing |

Options parameter is optional, so you need to specify only the name of the collection. Following is the list of options you can use −

|  |  |  |
| --- | --- | --- |
| **Field** | **Type** | **Description** |
| capped | Boolean | (Optional) If true, enables a capped collection. Capped collection is a fixed size collection that automatically overwrites its oldest entries when it reaches its maximum size. **If you specify true, you need to specify size parameter also.** |
| autoIndexId | Boolean | (Optional) If true, automatically create index on \_id field.s Default value is false. |
| size | number | (Optional) Specifies a maximum size in bytes for a capped collection. **If capped is true, then you need to specify this field also.** |
| max | number | (Optional) Specifies the maximum number of documents allowed in the capped collection. |

While inserting the document, MongoDB first checks size field of capped collection, then it checks max field.

### **Examples**

Basic syntax of **createCollection()** method without options is as follows −

>use flights

switched to db flights

>db.createCollection("indigo")

{ "ok" : 1 }

>

You can check the created collection by using the command **show collections**.

>show collections

indigo

The following example shows the syntax of **createCollection()** method with few important options −

> db.createCollection("AirAsia", { capped : true, autoIndexID : true, size : 6142800, max : 10000 }

## The drop() Method

MongoDB's **db.collection.drop()** is used to drop a collection from the database.

### **Syntax**

Basic syntax of **drop()** command is as follows −

db.COLLECTION\_NAME.drop()

### **Example**

First, check the available collections into your database **mydb**.

>use flights

switched to db flights

>show collections  
 indigo  
 AirAsia

>

Now drop the collection with the name **mycollection**.

>db.indigo.drop()

true

>

Again check the list of collections into database.

>show collections

AirAsia

>

drop() method will return true, if the selected collection is dropped successfully, otherwise it will return false.

**Mongo Datatypes**

* **String** − This is the most commonly used datatype to store the data. String in MongoDB must be UTF-8 valid.
* **Integer** − This type is used to store a numerical value. Integer can be 32 bit or 64 bit depending upon your server.
* **Boolean** − This type is used to store a boolean (true/ false) value.
* **Double** − This type is used to store floating point values.
* **Min/ Max keys** − This type is used to compare a value against the lowest and highest BSON elements.
* **Arrays** − This type is used to store arrays or list or multiple values into one key.
* **Timestamp**– current timestamp. This can be handy for recording when a document has been modified or added.
* **Object**− This datatype is used for embedded documents.
* **Null** − This type is used to store a Null value.
* **Symbol** − This datatype is used identically to a string; however, it's generally reserved for languages that use a specific symbol type.
* **Date** − This datatype is used to store the current date or time in UNIX time format. You can specify your own date time by creating object of Date and passing day, month, year into it.
* **Object ID** − This datatype is used to store the document’s ID.
* **Binary data** − This datatype is used to store binary data.
* **Code**− This datatype is used to store JavaScript code into the document.
* **Regular expression** − This datatype is used to store regular expression.

**Insert Document**  
To insert data into MongoDB collection, you need to use MongoDB's **insert()** or **save()** method.

### **Syntax**

The basic syntax of **insert()** command is as follows −

>db.COLLECTION\_NAME.insert(document)  
  
To insert the document you can use **db.post.save(document)** also. If you don't specify **\_id** in the document then **save()** method will work same as **insert()** method. If you specify \_id then it will replace whole data of document containing \_id as specified in save() method.  
Example:

db.firstbatch.insertMany([{

"o\_id":"abc123456789",

"name":"Prasad",

"mobile":8900989090,

"mail":"prasad@k7infotech.com",

"course":"FullStack"

},

{

"o\_id":"abc1234567\_1",

"name":"Tharun",

"mobile":8900989090,

"mail":"tharun@k7infotech.com",

"course":"FullStack"

}])

output  
{

acknowledged: true,

insertedIds: {

'0': ObjectId('671c7b93a253b9011805b012'),

'1': ObjectId('671c7b93a253b9011805b013')

}

db.firstbatch.insertOne({ "o\_id":"abc123456789",

"name":"Deepak",

"mobile":8900989090,

"mail":"deepak@k7infotech.com",

"course":"FullStack"

})

Output :

{

acknowledged: true,

insertedId: ObjectId('671c7c04a253b9011805b014')

}  
  
**Find():**  
If you want check how many documents inserted in the collections  
db.collectionName.find()  
{

\_id: ObjectId('671c7c04a253b9011805b014'),

o\_id: 'abc123456789',

name: 'Deepak',

mobile: 8900989090,

mail: 'deepak@k7infotech.com',

course: 'FullStack'

}  
{

\_id: ObjectId('671c7b93a253b9011805b013'),

o\_id: 'abc1234567\_1',

name: 'Tharun',

mobile: 8900989090,

mail: 'tharun@k7infotech.com',

course: 'FullStack'

}  
  
findOne():  
findone will retrieve the first document in the collection.  
syntax: db.collectionName.findOne()

db.firstbatch.findOne()

{

{

\_id: ObjectId('671c7b72a253b9011805b010'),

o\_id: 'abc123456789',

name: 'Prasad',

mobile: 8900989090,

mail: 'prasad@k7infotech.com',

course: 'FullStack'

### }

### 1. Basic Retrieval

#### Find All Documents

To retrieve all documents from a collection:

db.students.find();

This command returns all the documents in the students collection.

### 2. Filtering Results

#### Find with Conditions

You can filter results by specifying conditions in the find() method.

##### Example: Find Students with a Specific Age

To find students who are exactly 20 years old:

db.students.find({ age: 20 });

##### Example: Find Students by Major

To find all students majoring in "Computer Science":

db.students.find({ major: "Computer Science" });

### 3. Projections

You can specify which fields to include or exclude in the result set.

#### Example: Retrieve Only Specific Fields

To retrieve only the name and email fields of all documents:

db.students.find({}, { name: 1, email: 1 });

* The first {} represents the query filter (empty means all documents).
* The second { name: 1, email: 1 } specifies which fields to include (1 means include, 0 means exclude).

### 4. Advanced Queries

#### Using Comparison Operators

MongoDB supports several comparison operators:

* **Greater Than ($gt)**: To find students older than 20:

db.students.find({ age: { $gt: 20 } });

* **Less Than ($lt)**: To find students younger than 20:

db.students.find({ age: { $lt: 20 } });

* **Greater Than or Equal To ($gte)**: To find students aged 20 and above:

db.students.find({ age: { $gte: 20 } });

* **Less Than or Equal To ($lte)**: To find students aged 20 and below:

db.students.find({ age: { $lte: 20 } });

#### Logical Operators

You can combine multiple conditions using logical operators.

* **AND ($and)**: Find students who are 20 years old and majoring in "Computer Science":

db.students.find({ $and: [ { age: 20 }, { major: "Computer Science" } ] });

* **OR ($or)**: Find students who are either majoring in "Mathematics" or "Physics":

db.students.find({ $or: [ { major: "Mathematics" }, { major: "Physics" } ] });

* **NOT ($not)**: Find students who are not majoring in "Computer Science":

db.students.find({ major: { $not: { $eq: "Computer Science" } } });

* **NOTEQUAL($ne): find students not equal to “ComputerScience”**db.stuudents.find({major:{$ne:”Computer Science”}})

### 5. Regular Expressions

You can use regex to find documents with matching patterns.

#### Example: Find Students Whose Names Start with "J"

db.students.find({ name: /^J/ });

* The ^ indicates that the name should start with "J".
* db.secondbatch.find({"Address.DNo":/^\d+$/})

### 6. Sorting Results

You can sort the results using the sort() method.

#### Example: Sort by Age

To sort students by age in ascending order:

db.students.find().sort({ age: 1 }); // 1 for ascending

To sort by age in descending order:

db.students.find().sort({ age: -1 }); // -1 for descending

### 7. Limiting Results

You can limit the number of documents returned using the limit() method.

#### Example: Limit to 5 Results

To get only the first 5 student documents:

db.students.find().limit(5);

### 8. Combining Methods

You can combine these methods to refine your queries further.

#### Example: Find Students Aged 20, Sort by Name, Limit to 5

db.students.find({ age: 20 }).sort({ name: 1 }).limit(5);  
db.secondbatch.find({"age":{$gte:20}}).sort({"id":-1}).limit(10)

### 9. Using findOne()

If you want to find a single document, use findOne(). This method returns the first document that matches the query.

#### Example: Find One Student by Email

db.students.findOne({ email: "john.doe@example.com" });

### 10. Viewing Results

To format the output for better readability, you can use the pretty() method:

db.students.find().pretty();

This will display the documents in a nicely formatted way.

RDBMS Where Clause Equivalents in MongoDB

To query the document on the basis of some condition, you can use following operations.

|  |  |  |  |
| --- | --- | --- | --- |
| **Operation** | **Syntax** | **Example** | **RDBMS Equivalent** |
| Equality | {<key>:{$eg;<value>}} | db.mycol.find({"by":"tutorials point"}).pretty() | where by = 'tutorials point' |
| Less Than | {<key>:{$lt:<value>}} | db.mycol.find({"likes":{$lt:50}}).pretty() | where likes < 50 |
| Less Than Equals | {<key>:{$lte:<value>}} | db.mycol.find({"likes":{$lte:50}}).pretty() | where likes <= 50 |
| Greater Than | {<key>:{$gt:<value>}} | db.mycol.find({"likes":{$gt:50}}).pretty() | where likes > 50 |
| Greater Than Equals | {<key>:{$gte:<value>}} | db.mycol.find({"likes":{$gte:50}}).pretty() | where likes >= 50 |
| Not Equals | {<key>:{$ne:<value>}} | db.mycol.find({"likes":{$ne:50}}).pretty() | where likes != 50 |
| Values in an array | {<key>:{$in:[<value1>, <value2>,……<valueN>]}} | db.mycol.find({"name":{$in:["Raj", "Ram", "Raghu"]}}).pretty() | Where name matches any of the value in :["Raj", "Ram", "Raghu"] |
| Values not in an array | {<key>:{$nin:<value>}} | db.mycol.find({"name":{$nin:["Ramu", "Raghav"]}}).pretty() | Where name values is not in the array :["Ramu", "Raghav"] or, doesn’t exist at all |

### Update One Document

#### Using updateOne()

This method updates a single document that matches the specified filter.

**Syntax:**

db.collection.updateOne(

<filter>,

<update>,

{ options }

)

**Example: Update a Student's Email** To update the email of a student with a specific name:  
  
db.students.updateOne(

{ name: "John Doe" }, // Filter

{ $set: { email: "john.new@example.com" } } // Update

);  
  
  
  
db.secondbatch.updateMany({"age":{$gt: 24}},{$set :{"salary":25000}})

* The $set operator specifies the fields to update.

### 2. Update Multiple Documents

#### Using updateMany()

This method updates all documents that match the specified filter.

**Syntax:**

db.collection.updateMany(

<filter>,

<update>,

{ options }

)

**Example: Update All Students' Majors** To change the major of all students aged 20 to "Undeclared":

javascript

Copy code

db.students.updateMany(

{ age: 20 }, // Filter

{ $set: { major: "Undeclared" } } // Update

);  
db.secondbatch.updateMany({"createdBy":"System", "Address.City":"Tirupathi"},{$set:{"Address.Pincode":517501}})

db.secondbatch.updateMany({"experience":{$gt:2}},{$set:{"created\_at":Date()}})

### 3. Update with Multiple Operators

You can use multiple operators in a single update operation.

#### Example: Update Name and Age

To update both the name and age of a student:

db.students.updateOne(

{ email: "john.doe@example.com" }, // Filter

{

$set: { name: "Johnathan Doe" }, // Update name

$inc: { age: 1 } // Increment age by 1

}

);

db.secondbatch.updateMany(

{"createdBy": "System"},

{$set :{"Organisation":"K7Info Tech Pvt Ltd"},

$inc:{"age":1}

})

output:  
{

acknowledged: true,

insertedId: null,

matchedCount: 3,

modifiedCount: 3,

upsertedCount: 0

}

Example 2:   
for decrement , there is no decrement keyword, we need to use $inc keyword with negative values (-1)  
  
db.secondbatch.updateMany(

{},

{$set :{"Organisation":"K7InfoTech Pvt Ltd"},

$inc:{"age":-2}

})

### 4. Upsert Operation

An "upsert" operation creates a new document if no document matches the filter.

#### Example: Upsert a Student

To insert a new student if they do not exist, or update their email if they do:

db.students.updateOne(

{ name: "Jane Doe" }, // Filter

{ $set: { email: "jane.new@example.com" } }, // Update

{ upsert: true } // Upsert option

);  
  
Ex:

db.secondbatch.updateMany(

{"age":29},

{$set :{"experince":3},

$inc:{"age":-2}

},{"upsert":true})

### 5. Find and Update

#### Using findOneAndUpdate()

This method finds a single document and updates it in one atomic operation. It returns the original document or the updated document.

**Syntax:**

db.collection.findOneAndUpdate(

<filter>,

<update>,

{ options }

)

**Example: Find and Update a Student's Age** To find a student by email and update their age:

db.students.findOneAndUpdate(

{ email: "jane.smith@example.com" }, // Filter

{ $set: { age: 21 } }, // Update

{ returnDocument: "after" } // Return the updated document

);

### 6. Remove Fields

To remove a field from a document, you can use the $unset operator.

#### Example: Remove a Student's Major

To remove the major field from a student document:

db.students.updateOne(

{ name: "John Doe" }, // Filter

{ $unset: { major: "" } } // Remove major field

);  
db.secondbatch.updateMany({"name":"Prasad"},{$unset :{"employee":""}})

### Update Operators

* **$set**: Sets the value of a field in a document.
* **$unset**: Removes a field from a document.
* **$inc**: Increments a field by a specified value.
* **$push**: Adds an item to an array.
* **$addToSet**: Adds an item to an array only if it doesn’t already exist.
* **$pull**: Removes an item from an array.

### $inc: Increments a Field by a Specified Value

The $inc operator is used to increase (or decrease) the value of a field by a specified amount.

**Example: Incrementing a Student's Age** Suppose you have a student document with the following structure:

{

"\_id": 1,

"name": "John Doe",

"age": 20,

"credits": 30

}

To increment John's age by 1:

db.students.updateOne(

{ name: "John Doe" }, // Filter

{ $inc: { age: 1 } } // Increment age by 1

);

To increment John's credits by 15:

db.students.updateOne(

{ name: "John Doe" }, // Filter

{ $inc: { credits: 15 } } // Increment credits by 15

);

### 2. $push: Adds an Item to an Array

The $push operator adds a specified value to an array.

**Example: Adding a Course to a Student's List** Assuming each student has a courses array:

{

"\_id": 1,

"name": "John Doe",

"courses": ["Math", "Science"]

}

To add a new course, "History":

db.students.updateOne(

{ name: "John Doe" }, // Filter

{ $push: { courses: "History" } } // Add "History" to the courses array

);

db.secondbatch.updateMany({"experience":{$lte:5}},{$push:{"course":["DevOps","FullStack"]}})

After the operation, John's document would look like:

{

"\_id": 1,

"name": "John Doe",

"courses": ["Math", "Science", "History"]

}

### 3. $addToSet: Adds an Item to an Array Only If It Doesn’t Already Exist

The $addToSet operator ensures that a value is added to an array only if it is not already present.

**Example: Adding a Unique Course** Using the same student document:

{

"\_id": 1,

"name": "John Doe",

"courses": ["Math", "Science"]

}

To add a course, "Math", which already exists:

db.students.updateOne(

{ name: "John Doe" }, // Filter

{ $addToSet: { courses: "Math" } } // Attempt to add "Math"

);

The courses array remains unchanged:

{

"\_id": 1,

"name": "John Doe",

"courses": ["Math", "Science"]

}

To add a new course, "English":

db.students.updateOne(

{ name: "John Doe" }, // Filter

{ $addToSet: { courses: "English" } } // Add "English" to the courses array

);

db.secondbatch.updateMany({"experience":{$lte:5}},{$addToSet:{"course":"FullStack"}})

After the operation, John's document would look like:

{

"\_id": 1,

"name": "John Doe",

"courses": ["Math", "Science", "English"]

}

### 4. $pull: Removes an Item from an Array

The $pull operator removes all instances of a value from an array.

**Example: Removing a Course from a Student's List** Continuing with the previous student document:

{

"\_id": 1,

"name": "John Doe",

"courses": ["Math", "Science", "History", "English"]

}

To remove the course "History":

db.students.updateOne(

{ name: "John Doe" }, // Filter

{ $pull: { courses: "History" } } // Remove "History" from the courses array

);

db.secondbatch.updateMany({"experience":{$lte:5}},{$pull:{"course":"DevOps"}})

After the operation, John's document would look like:

{

"\_id": 1,

"name": "John Doe",

"courses": ["Math", "Science", "English"]

}