MongoDB

Introduction

MongoDB is an open-source document database and leading NoSQL database. MongoDB is written in C++. This tutorial will give you great understanding on MongoDB concepts needed to create and deploy a highly scalable and performance-oriented database.

MongoDB is a cross-platform, document oriented database that provides, high performance, high availability, and easy scalability. MongoDB works on concept of collection and document.

Database

Database is a physical container for collections. Each database gets its own set of files on the file system. A single MongoDB server typically has multiple databases.

Collection

Collection is a group of MongoDB documents. It is the equivalent of an RDBMS table. A collection exists within a single database. Collections do not enforce a schema. Documents within a collection can have different fields. Typically, all documents in a collection are of similar or related purpose.

Document

A document is a set of key-value pairs. Documents have dynamic schema. Dynamic schema means that documents in the same collection do not need to have the same set of fields or structure, and common fields in a collection's documents may hold different types of data.

The following table shows the relationship of RDBMS terminology with MongoDB.

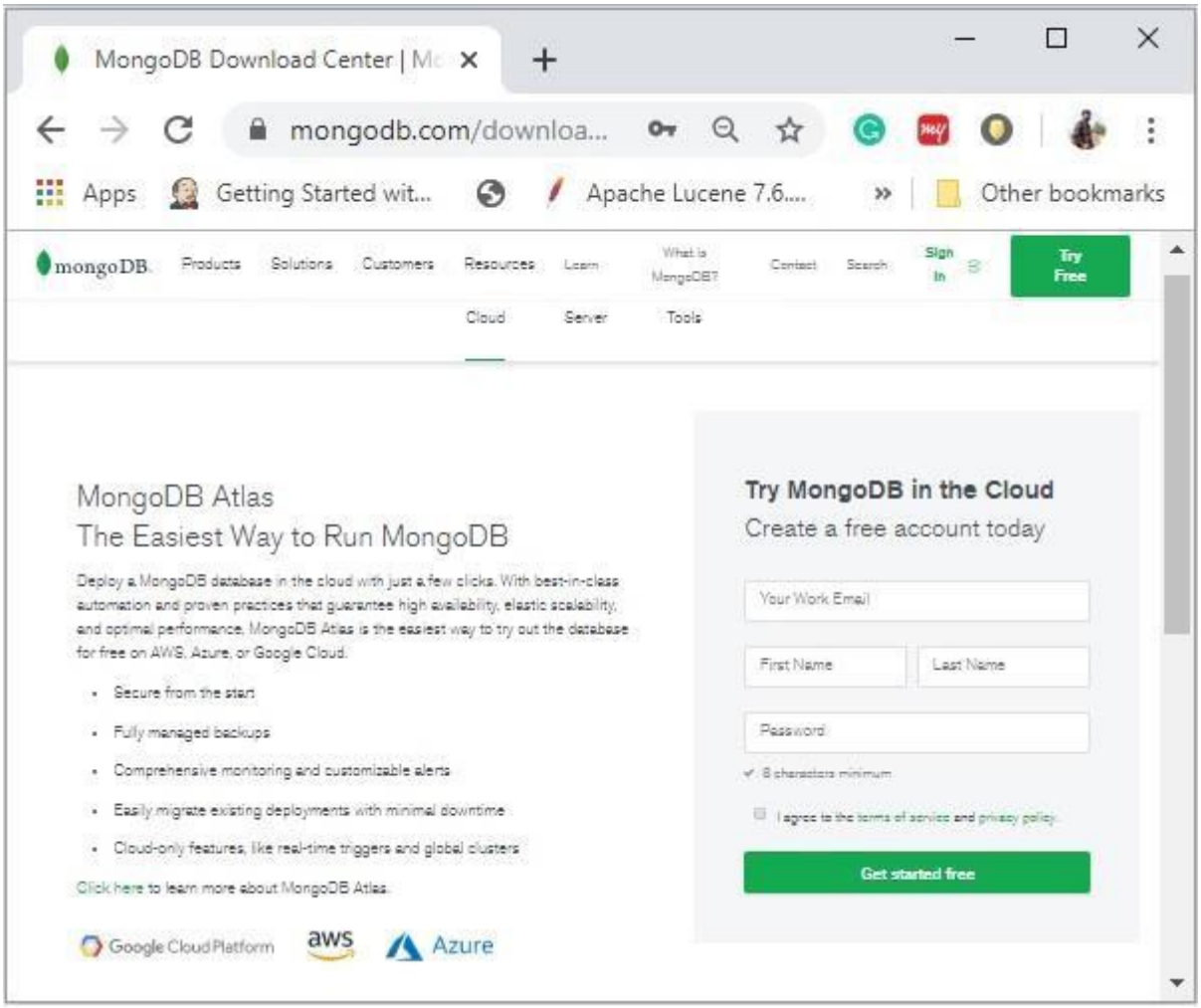
RDBMS	MongoDB
Database	Database
Table	Collection
Tuple/Row	Document
column	Field
Primary Key	Primary Key (Default key _id provided by MongoDB itself)

MongoDB - Environment

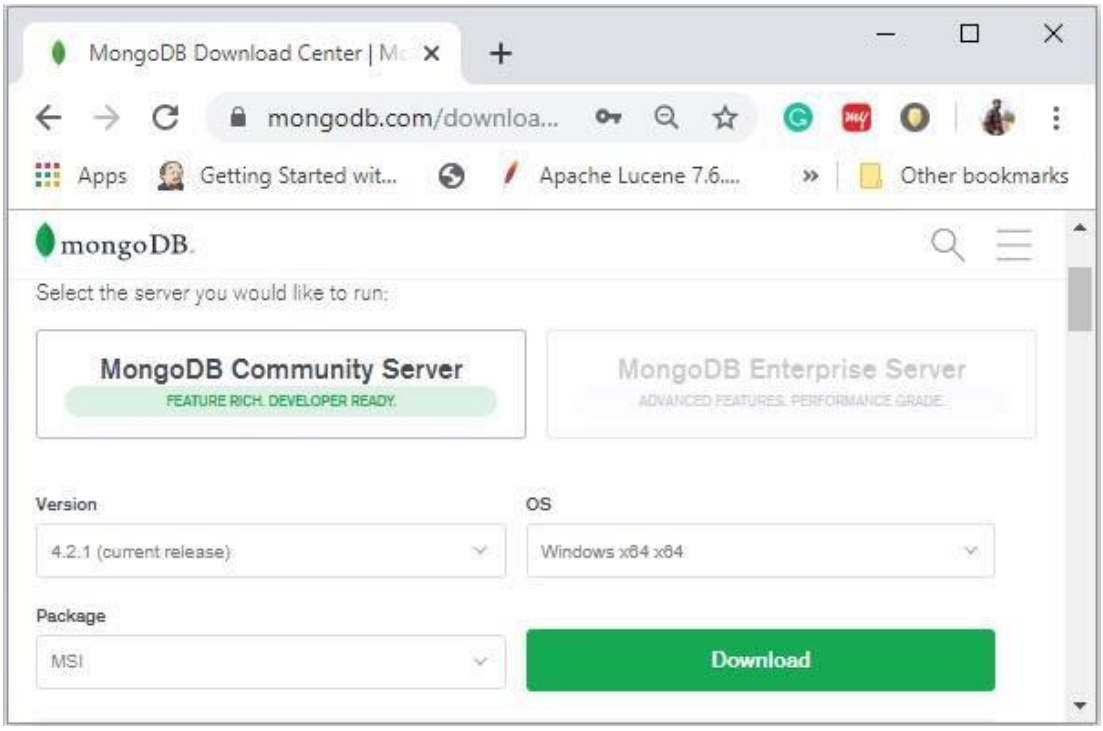
Let us now see how to install MongoDB on Windows.

Install MongoDB On Windows

To install MongoDB on Windows, first download the latest release of MongoDB from https://www.mongodb.com/download-center.



Enter the required details, select the ***Server*** tab, in it you can choose the version of MongoDB, operating system and, packaging as:



Now install the downloaded file, by default, it will be installed in the folder **C:\Program Files**.

MongoDB requires a data folder to store its files. The default location for the MongoDB data directory is c:\data\db. So you need to create this folder using the Command Prompt. Execute the following command sequence.

```
C:\>md data
C:\md data\db
```

Then you need to specify set the **dbpath** to the created directory in **mongod.exe**. For the same, issue the following commands.

In the command prompt, navigate to the bin directory current in the MongoDB installation folder. Suppose my installation folder is **C:\Program Files\MongoDB**

```
C:\Users\XYZ>d:cd C:\Program Files\MongoDB\Server\4.2\bin
C:\Program Files\MongoDB\Server\4.2\bin>mongod.exe --dbpath "C:\data"
```

This will show **waiting for connections** message on the console output, which indicates that the mongod.exe process is running successfully.

Now to run the MongoDB, you need to open another command prompt and issue the following command.

```
C:\Program Files\MongoDB\Server\4.2\bin>mongo.exe
MongoDB shell version v4.2.1
connecting to:
mongodb://127.0.0.1:27017/?compressors=disabled&gssapiServiceName=mongodb
Implicit session: session { "id" : UUID("4260beda-f662-4cbe-9bc7-5c1f2242663c") }
MongoDB server version: 4.2.1
>
```

This will show that MongoDB is installed and run successfully. Next time when you run MongoDB, you need to issue only commands.

```
C:\Program Files\MongoDB\Server\4.2\bin>mongod.exe --dbpath "C:\data"
C:\Program Files\MongoDB\Server\4.2\bin>mongo.exe
```

MongoDB - Data Modelling

Data in MongoDB has a flexible schema.documents in the same collection. They do not need to have the same set of fields or structure Common fields in a collection's documents may hold different types of data.

Data Model Design

MongoDB provides two types of data models: — Embedded data model and Normalized data model. Based on the requirement, you can use either of the models while preparing your document.

Embedded Data Model

In this model, you can have (embed) all the related data in a single document, it is also known as de-normalized data model.

For example, assume we are getting the details of employees in three different documents namely, Personal_details, Contact and, Address, you can embed all the three documents in a single one as shown below –

```
{
        _id: ,
        Emp_ID: "10025AE336"
        Personal_details:{
                First_Name: "Ravindra",
                Last_Name: "Sharma",
                Date_Of_Birth: "1995-09-26"
        },
        Contact: {
                e-mail: "Ravindra_sharma.123@gmail.com",
                phone: "9848022338"
        },
        Address: {
                city: "Hyderabad",
                Area: "Madapur",
                State: "Telangana"
        }
}
```

Normalized Data Model

In this model, you can refer the sub documents in the original document, using references. For example, you can re-write the above document in the normalized model as:

Employee:

```
{
        _id: <ObjectId101>,
        Emp_ID: "10025AE336"
}
```

Personal_details:

```
{
        _id: <ObjectId102>,
        empDocID: " ObjectId101",
        First_Name: "Ravindra",
        Last_Name: "Sharma",
        Date_Of_Birth: "1995-09-26"
}
```

Contact:

```
{
        _id: <ObjectId103>,
        empDocID: " ObjectId101",
        e-mail: "Ravindra_sharma.123@gmail.com",
        phone: "9848022338"
}
```

Address:

```
{
        _id: <ObjectId104>,
        empDocID: " ObjectId101",
        city: "Hyderabad",
        Area: "Madapur",
        State: "Telangana"
}
```

Considerations while designing Schema in MongoDB

- Design your schema according to user requirements.
- Combine objects into one document if you will use them together. Otherwise separate them (but make sure there should not be need of joins).

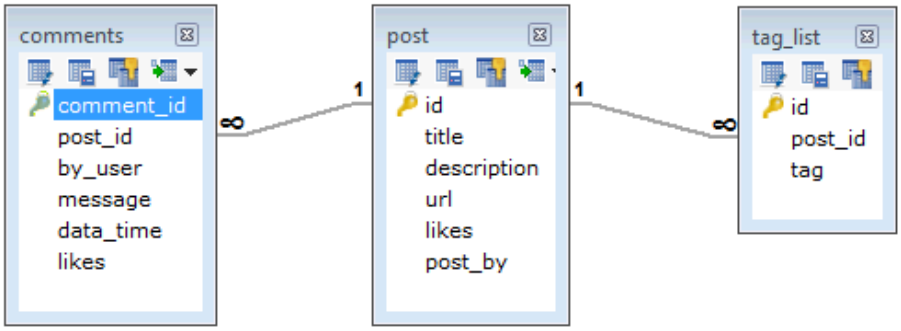
- Duplicate the data (but limited) because disk space is cheap as compare to compute time.
- Do joins while write, not on read.
- Optimize your schema for most frequent use cases.
- Do complex aggregation in the schema.

Example

Suppose a client needs a database design for his blog/website and see the differences between RDBMS and MongoDB schema design. Website has the following requirements.

- Every post has the unique title, description and url.
- Every post can have one or more tags.
- Every post has the name of its publisher and total number of likes.
- Every post has comments given by users along with their name, message, data-time and likes.
- On each post, there can be zero or more comments.

In RDBMS schema, design for above requirements will have minimum three tables.



While in MongoDB schema, design will have one collection post and this structure –

```
{
  _id: POST_ID
  title: TITLE_OF_POST,
  description: POST_DESCRIPTION,
  by: POST_BY,
  url: URL_OF_POST,
  tags: [TAG1, TAG2, TAG3],
  likes: TOTAL_LIKES,
  comments: [
    {
      user:'COMMENT_BY',
      message: TEXT,
      dateCreated: DATE_TIME,
      like: LIKES
    },
    {
```

```
      user:'COMMENT_BY',
      message: TEXT,
      dateCreated: DATE_TIME,
      like: LIKES
    }
  ]
}
```

So while showing the data, in RDBMS you need to join three tables and in MongoDB, data will be shown from one collection only.

MongoDB - Create Database

In this chapter, we will see how to create a database in MongoDB.

The use Command

MongoDB **use DATABASE_NAME** is used to create database. The command will create a new database if it doesn't exist, otherwise it will return the existing database.

Syntax

Basic syntax of **use DATABASE** statement is as follows –

```
use DATABASE_NAME
```

Example

If you want to use a database with name **<mydb>**, then **use DATABASE** statement would be as follows –

```
>use mydb
switched to db mydb
```

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

```
>db
mydb
```

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

```
>show dbs
local    0.78125GB
test     0.23012GB
```

Your created database (mydb) is not present in list. To display database, you need to insert at least one document into it.

```
>db.movie.insert({"name":"sudaksha"})
>show dbs
local     0.78125GB
mydb      0.23012GB
test      0.23012GB
```

In MongoDB default database is test. If you didn't create any database, then collections will be stored in test database.

MongoDB - Create Collection

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 test
switched to db test
>db.createCollection("mycollection")
{ "ok" : 1 }
>
```

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

```
>show collections
mycollection
system.indexes
```

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

```
> db.createCollection("mycol", { capped : true, autoIndexID : true, size : 6142800, max :
10000 } ){
"ok" : 0,
"errmsg" : "BSON field 'create.autoIndexID' is an unknown field.",
"code" : 40415,
"codeName" : "Location40415"
}
>
```

In MongoDB, you don't need to create collection. MongoDB creates collection automatically, when you insert some document.

```
>db.tutorialspoint.insert({"name" : "tutorialspoint"}),
WriteResult({ "nInserted" : 1 })
>show collections
mycol
mycollection
system.indexes
tutorialspoint
>
```

MongoDB - Datatypes

MongoDB supports many datatypes. Some of them are –

- **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** – ctimestamp. 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.

MongoDB - Insert Document

The insert() Method

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)
```

Example

```
> db.users.insert({
... _id : ObjectId("507f191e810c19729de860ea"),
... title: "MongoDB Overview",
... description: "MongoDB is no sql database",
... by: "sudaksha",
... url: "http://www.tutorialspoint.com",
... tags: ['mongodb', 'database', 'NoSQL'],
... likes: 100
... })
WriteResult({ "nInserted" : 1 })
>
```

Here **mycol** is our collection name, as created in the previous chapter. If the collection doesn't exist in the database, then MongoDB will create this collection and then insert a document into it.

In the inserted document, if we don't specify the _id parameter, then MongoDB assigns a unique ObjectId for this document.

_id is 12 bytes hexadecimal number unique for every document in a collection. 12 bytes are divided as follows –

```
_id: ObjectId(4 bytes timestamp, 3 bytes machine id, 2 bytes process id, 3 bytes incrementer)
```

You can also pass an array of documents into the insert() method as shown below:.

```
> db.createCollection("post")
> db.post.insert([
        {
                title: "MongoDB Overview",
                description: "MongoDB is no SQL database",
                by: "sudaksha",
                url: "http://www.tutorialspoint.com",
                tags: ["mongodb", "database", "NoSQL"],
                likes: 100
```

```
        },
        {
        title: "NoSQL Database",
        description: "NoSQL database doesn't have tables",
        by: "sudaksha",
        url: "http://www.tutorialspoint.com",
        tags: ["mongodb", "database", "NoSQL"],
        likes: 20,
        comments: [
                {
                        user:"user1",
                        message: "My first comment",
                        dateCreated: new Date(2013,11,10,2,35),
                        like: 0
                }
        ]
}
])
BulkWriteResult({
        "writeErrors" : [ ],
        "writeConcernErrors" : [ ],
        "nInserted" : 2,
        "nUpserted" : 0,
        "nMatched" : 0,
        "nModified" : 0,
        "nRemoved" : 0,
        "upserted" : [ ]
})
>
```

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.

The insertOne() method

If you need to insert only one document into a collection you can use this method.

Syntax

The basic syntax of insert() command is as follows –

```
>db.COLLECTION_NAME.insertOne(document)
```

Example

Following example creates a new collection named empDetails and inserts a document using the insertOne() method.

```
> db.createCollection("empDetails")
{ "ok" : 1 }
> db.empDetails.insertOne(
        {
                First_Name: "Ravindra",
                Last_Name: "Sharma",
                Date_Of_Birth: "1995-09-26",
                e_mail: "Ravindra_sharma.123@gmail.com",
                phone: "9848022338"
        })
{
        "acknowledged" : true,
        "insertedId" : ObjectId("5dd62b4070fb13eec3963bea")
}
>
```

The insertMany() method

You can insert multiple documents using the insertMany() method. To this method you need to pass an array of documents.

Example

Following example inserts three different documents into the empDetails collection using the insertMany() method.

```
> db.empDetails.insertMany(
        [
                {
                        First_Name: "Ravindra",
                        Last_Name: "Sharma",
                        Date_Of_Birth: "1995-09-26",
                        e_mail: "Ravindra_sharma.123@gmail.com",
                        phone: "9000012345"
                },
                {
                        First_Name: "Evelyn",
                        Last_Name: "Benedict",
                        Date_Of_Birth: "1990-02-16",
                        e_mail: "Evelyn_Benedict.123@gmail.com",
                        phone: "9000054321"
                },
                {
                        First_Name: "Yasmin",
                        Last_Name: "Sheik",
                        Date_Of_Birth: "1990-02-16",
                        e_mail: "Yasmin_Sheik.123@gmail.com",
                        phone: "9000054321"
                }
        ]
```

```
)
{
        "acknowledged" : true,
        "insertedIds" : [
                ObjectId("5dd631f270fb13eec3963bed"),
                ObjectId("5dd631f270fb13eec3963bee"),
                ObjectId("5dd631f270fb13eec3963bef")
        ]
}
>
```

MongoDB - Query Document

The find() Method

To query data from MongoDB collection, you need to use MongoDB's **find()** method.

Syntax

The basic syntax of **find()** method is as follows –

```
>db.COLLECTION_NAME.find()
```

find() method will display all the documents in a non-structured way.

Example

Assume we have created a collection named mycol as –

```
> use sampleDB
switched to db sampleDB
> db.createCollection("mycol")
{ "ok" : 1 }
>
```

And inserted 3 documents in it using the insert() method as shown below –

```
> db.mycol.insert([
        {
                title: "MongoDB Overview",
                description: "MongoDB is no SQL database",
                by: "sudaksha",
                url: "http://www.tutorialspoint.com",
                tags: ["mongodb", "database", "NoSQL"],
                likes: 100
        },
        {
                title: "NoSQL Database",
                description: "NoSQL database doesn't have tables",
                by: "sudaksha",
```

```
                    url: "http://www.tutorialspoint.com",
                    tags: ["mongodb", "database", "NoSQL"],
                    likes: 20,
                    comments: [
                            {
                                    user:"user1",
                                    message: "My first comment",
                                    dateCreated: new Date(2013,11,10,2,35),
                                    like: 0
                            }
                    ]
            }
])
```

Following method retrieves all the documents in the collection –

```
> db.mycol.find()
{ "_id" : ObjectId("5dd4e2cc0821d3b44607534c"), "title" : "MongoDB Overview",
"description" : "MongoDB is no SQL database", "by" : "sudaksha", "url" :
"http://www.tutorialspoint.com", "tags" : [ "mongodb", "database", "NoSQL" ], "likes" : 100 }
{ "_id" : ObjectId("5dd4e2cc0821d3b44607534d"), "title" : "NoSQL Database", "description"
: "NoSQL database doesn't have tables", "by" : "sudaksha", "url" :
"http://www.tutorialspoint.com", "tags" : [ "mongodb", "database", "NoSQL" ], "likes" : 20,
"comments" : [ { "user" : "user1", "message" : "My first comment", "dateCreated" :
ISODate("2013-12-09T21:05:00Z"), "like" : 0 } ] }
>
```

The pretty() Method

To display the results in a formatted way, you can use pretty() method.

Syntax

```
>db.COLLECTION_NAME.find().pretty()
```

Example

Following example retrieves all the documents from the collection named mycol and arranges them in an easy-to-read format.

```
> db.mycol.find().pretty()
{
        "_id" : ObjectId("5dd4e2cc0821d3b44607534c"),
        "title" : "MongoDB Overview",
        "description" : "MongoDB is no SQL database",
        "by" : "sudaksha",
        "url" : "http://www.tutorialspoint.com",
        "tags" : [
                "mongodb",
                "database",
                "NoSQL"
        ],
```

```
        "likes" : 100
}
{
        "_id" : ObjectId("5dd4e2cc0821d3b44607534d"),
        "title" : "NoSQL Database",
        "description" : "NoSQL database doesn't have tables",
        "by" : "sudaksha",
        "url" : "http://www.tutorialspoint.com",
        "tags" : [
                "mongodb",
                "database",
                "NoSQL"
        ],
        "likes" : 20,
        "comments" : [
                {
                        "user" : "user1",
                        "message" : "My first comment",
                        "dateCreated" : ISODate("2013-12-09T21:05:00Z"),
                        "like" : 0
                }
        ]
}
```

The findOne() method

Apart from the find() method, there is **findOne()** method, that returns only one document.

Syntax

```
>db.COLLECTIONNAME.findOne()
```

Example

Following example retrieves the document with title MongoDB Overview.

```
> db.mycol.findOne({title: "MongoDB Overview"})
{
        "_id" : ObjectId("5dd6542170fb13eec3963bf0"),
        "title" : "MongoDB Overview",
        "description" : "MongoDB is no SQL database",
        "by" : "sudaksha",
        "url" : "http://www.tutorialspoint.com",
        "tags" : [
                "mongodb",
                "database",
                "NoSQL"
        ],
        "likes" : 100
}
```

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>:{$eg;<value>}}	db.mycol.find({"by":"sudaksha"}).pretty()	where by = 'sudaksha'
Less Than	{<key>:{$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

AND in MongoDB

Syntax

To query documents based on the AND condition, you need to use $and keyword. Following is the basic syntax of AND –

```
>db.mycol.find({ $and: [ {<key1>:<value1>}, { <key2>:<value2>} ] })
```

Example

Following example will show all the tutorials written by 'sudaksha' and whose title is 'MongoDB Overview'.

```
> db.mycol.find({$and:[{"by":"sudaksha"},{"title": "MongoDB Overview"}]}).pretty()
{
        "_id" : ObjectId("5dd4e2cc0821d3b44607534c"),
        "title" : "MongoDB Overview",
        "description" : "MongoDB is no SQL database",
        "by" : "sudaksha",
        "url" : "http://www.tutorialspoint.com",
        "tags" : [
                "mongodb",
                "database",
                "NoSQL"
        ],
        "likes" : 100
}
>
```

For the above given example, equivalent where clause will be **' where by = 'sudaksha' AND title = 'MongoDB Overview' '**. You can pass any number of key, value pairs in find clause.

OR in MongoDB

Syntax

To query documents based on the OR condition, you need to use **$or** keyword. Following is the basic syntax of **OR** –

```
>db.mycol.find(
  {
    $or: [
      {key1: value1}, {key2:value2}
    ]
  }
).pretty()
```

Example

Following example will show all the tutorials written by 'sudaksha' or whose title is 'MongoDB Overview'.

```
>db.mycol.find({$or:[{"by":"sudaksha"},{"title": "MongoDB Overview"}]}).pretty()
{
  "_id": ObjectId(7df78ad8902c),
  "title": "MongoDB Overview",
  "description": "MongoDB is no sql database",
  "by": "sudaksha",
  "url": "http://www.tutorialspoint.com",
  "tags": ["mongodb", "database", "NoSQL"],
  "likes": "100"
}
>
```

Using AND and OR Together

Example

The following example will show the documents that have likes greater than 10 and whose title is either 'MongoDB Overview' or by is 'sudaksha'. Equivalent SQL where clause is **'where likes>10 AND (by = 'sudaksha' OR title = 'MongoDB Overview')'**

```
>db.mycol.find({"likes": {$gt:10}, $or: [{"by": "sudaksha"},
  {"title": "MongoDB Overview"}]}).pretty()
{
  "_id": ObjectId(7df78ad8902c),
  "title": "MongoDB Overview",
  "description": "MongoDB is no sql database",
  "by": "sudaksha",
  "url": "http://www.tutorialspoint.com",
  "tags": ["mongodb", "database", "NoSQL"],
```

```
  "likes": "100"
}
>
```

NOR in MongoDB

Syntax

To query documents based on the NOT condition, you need to use $not keyword. Following is the basic syntax of **NOT** –

```
>db.COLLECTION_NAME.find(
        {
                $not: [
                        {key1: value1}, {key2:value2}
                ]
        }
)
```

Example

Assume we have inserted 3 documents in the collection **empDetails** as shown below –

```
db.empDetails.insertMany(
        [
                {
                        First_Name: "Ravindra",
                        Last_Name: "Sharma",
                        Age: "26",
                        e_mail: "Ravindra_sharma.123@gmail.com",
                        phone: "9000012345"
                },
                {
                        First_Name: "Evelyn",
                        Last_Name: "Benedict",
                        Age: "27",
                        e_mail: "Evelyn_Benedict.123@gmail.com",
                        phone: "9000054321"
                },
                {
                        First_Name: "Yasmin",
                        Last_Name: "Sheik",
                        Age: "24",
                        e_mail: "Yasmin_Sheik.123@gmail.com",
                        phone: "9000054321"
                }
        ]
)
```

Following example will retrieve the document(s) whose first name is not "Ravindra" and last name is not "Benedict"

```
> db.empDetails.find(
        {
                $nor:[
                        40
                        {"First_Name": "Ravindra"},
                        {"Last_Name": "Benedict"}
                ]
        }
).pretty()
{
        "_id" : ObjectId("5dd631f270fb13eec3963bef"),
        "First_Name" : "Yasmin",
        "Last_Name" : "Sheik",
        "Age" : "24",
        "e_mail" : "Yasmin_Sheik.123@gmail.com",
        "phone" : "9000054321"
}
```

NOT in MongoDB

Syntax

To query documents based on the NOT condition, you need to use $not keyword following is the basic syntax of **NOT** –

```
>db.COLLECTION_NAME.find(
        {
                $NOT: [
                        {key1: value1}, {key2:value2}
                ]
        }
).pretty()
```

Example

Following example will retrieve the document(s) whose age is not greater than 25

```
> db.empDetails.find( { "Age": { $not: { $gt: "25" } } } )
{
        "_id" : ObjectId("5dd6636870fb13eec3963bf7"),
        "First_Name" : "Yasmin",
        "Last_Name" : "Sheik",
        "Age" : "24",
        "e_mail" : "Yasmin_Sheik.123@gmail.com",
        "phone" : "9000054321"
}
```

MongoDB - Update Document

MongoDB's **update()** and **save()** methods are used to update document into a collection. The update() method updates the values in the existing document while the save() method replaces the existing document with the document passed in save() method.

MongoDB Update() Method

The update() method updates the values in the existing document.

Syntax

The basic syntax of **update()** method is as follows –

```
>db.COLLECTION_NAME.update(SELECTION_CRITERIA, UPDATED_DATA)
```

Example

Consider the mycol collection has the following data.

```
{ "_id" : ObjectId(5983548781331adf45ec5), "title":"MongoDB Overview"}
{ "_id" : ObjectId(5983548781331adf45ec6), "title":"NoSQL Overview"}
{ "_id" : ObjectId(5983548781331adf45ec7), "title":"Sudaksha Overview"}
```

Following example will set the new title 'New MongoDB Tutorial' of the documents whose title is 'MongoDB Overview'.

```
>db.mycol.update({'title':'MongoDB Overview'},{$set:{'title':'New MongoDB Tutorial'}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
>db.mycol.find()
{ "_id" : ObjectId(5983548781331adf45ec5), "title":"New MongoDB Tutorial"}
{ "_id" : ObjectId(5983548781331adf45ec6), "title":"NoSQL Overview"}
{ "_id" : ObjectId(5983548781331adf45ec7), "title":"Sudaksha Overview"}
>
```

By default, MongoDB will update only a single document. To update multiple documents, you need to set a parameter 'multi' to true.

```
>db.mycol.update({'title':'MongoDB Overview'},
  {$set:{'title':'New MongoDB Tutorial'}},{multi:true})
```

MongoDB Save() Method

The **save()** method replaces the existing document with the new document passed in the save() method.

Syntax

The basic syntax of MongoDB **save()** method is shown below –

```
>db.COLLECTION_NAME.save({_id:ObjectId(),NEW_DATA})
```

Example

Following example will replace the document with the _id '5983548781331adf45ec5'.

```
>db.mycol.save(
  {
    "_id" : ObjectId("507f191e810c19729de860ea"),
                        "title":"Sudaksha New Topic",
    "by":"Sudaksha"
  }
)
WriteResult({
            "nMatched" : 0,
            "nUpserted" : 1,
            "nModified" : 0,
            "_id" : ObjectId("507f191e810c19729de860ea")
})
>db.mycol.find()
{ "_id" : ObjectId("507f191e810c19729de860e6"), "title":"Sudaksha New Topic",
  "by":"Sudaksha"}
{ "_id" : ObjectId("507f191e810c19729de860e6"), "title":"NoSQL Overview"}
{ "_id" : ObjectId("507f191e810c19729de860e6"), "title":"Sudaksha Overview"}
>
```

MongoDB findOneAndUpdate() method

The **findOneAndUpdate()** method updates the values in the existing document.

Syntax

The basic syntax of **findOneAndUpdate()** method is as follows –

```
>db.COLLECTION_NAME.findOneAndUpdate(SELECTIOIN_CRITERIA, UPDATED_DATA)
```

Example

Assume we have created a collection named empDetails and inserted three documents in it as shown below –

```
> db.empDetails.insertMany(
            [
                        {
                                    First_Name: "Ravindra",
                                    Last_Name: "Sharma",
                                    Age: "26",
                                    e_mail: "Ravindra_sharma.123@gmail.com",
                                    phone: "9000012345"
```

```
                },
                {
                        First_Name: "Evelyn",
                        Last_Name: "Benedict",
                        Age: "27",
                        e_mail: "Evelyn_Benedict.123@gmail.com",
                        phone: "9000054321"
                },
                {
                        First_Name: "Yasmin",
                        Last_Name: "Sheik",
                        Age: "24",
                        e_mail: "Yasmin_Sheik.123@gmail.com",
                        phone: "9000054321"
                }
        ]
)
```

Following example updates the age and email values of the document with name 'Ravindra'.

```
> db.empDetails.findOneAndUpdate(
        {First_Name: 'Ravindra'},
        { $set: { Age: '30',e_mail: 'Ravindra_newemail@gmail.com'}}
)
{
        "_id" : ObjectId("5dd6636870fb13eec3963bf5"),
        "First_Name" : "Ravindra",
        "Last_Name" : "Sharma",
        "Age" : "30",
        "e_mail" : "Ravindra_newemail@gmail.com",
        "phone" : "9000012345"
}
```

MongoDB updateOne() method

This methods updates a single document which matches the given filter.

Syntax

The basic syntax of updateOne() method is as follows –

```
>db.COLLECTION_NAME.updateOne(<filter>, <update>)
```

Example

```
> db.empDetails.updateOne(
        {First_Name: 'Ravindra'},
        { $set: { Age: '30',e_mail: 'Ravindra_newemail@gmail.com'}}
)
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 0 }
```

```
>
```

MongoDB updateMany() method

The updateMany() method updates all the documents that matches the given filter.

Syntax

The basic syntax of updateMany() method is as follows –

```
>db.COLLECTION_NAME.update(<filter>, <update>)
```

Example

```
> db.empDetails.updateMany(
            {Age:{ $gt: "25" }},
            { $set: { Age: '00'}}
)
{ "acknowledged" : true, "matchedCount" : 2, "modifiedCount" : 2 }
```

You can see the updated values if you retrieve the contents of the document using the find method as shown below –

```
> db.empDetails.find()
{ "_id" : ObjectId("5dd6636870fb13eec3963bf5"), "First_Name" : "Ravindra", "Last_Name" :
"Sharma", "Age" : "00", "e_mail" : "Ravindra_newemail@gmail.com", "phone" :
"9000012345" }
{ "_id" : ObjectId("5dd6636870fb13eec3963bf6"), "First_Name" : "Evelyn", "Last_Name" :
"Benedict", "Age" : "00", "e_mail" : "Evelyn_Benedict.123@gmail.com", "phone" :
"9000054321" }
{ "_id" : ObjectId("5dd6636870fb13eec3963bf7"), "First_Name" : "Yasmin", "Last_Name" :
"Sheik", "Age" : "24", "e_mail" : "Yasmin_Sheik.123@gmail.com", "phone" : "9000054321" }
>
```

MongoDB - Delete Document

The remove() Method

MongoDB's **remove()** method is used to remove a document from the collection. remove() method accepts two parameters. One is deletion criteria and second is justOne flag.

- **deletion criteria** – (Optional) deletion criteria according to documents will be removed.
- **justOne** – (Optional) if set to true or 1, then remove only one document.

Syntax

Basic syntax of **remove()** method is as follows –

```
>db.COLLECTION_NAME.remove(DELLETION_CRITTERIA)
```

Example

Consider the mycol collection has the following data.

```
{_id : ObjectId("507f191e810c19729de860e1"), title: "MongoDB Overview"},
{_id : ObjectId("507f191e810c19729de860e2"), title: "NoSQL Overview"},
{_id : ObjectId("507f191e810c19729de860e3"), title: "Sudaksha Overview"}
```

Following example will remove all the documents whose title is 'MongoDB Overview'.

```
>db.mycol.remove({'title':'MongoDB Overview'})
WriteResult({"nRemoved" : 1})
> db.mycol.find()
{"_id" : ObjectId("507f191e810c19729de860e2"), "title" : "NoSQL Overview" }
{"_id" : ObjectId("507f191e810c19729de860e3"), "title" : "Sudaksha Overview" }
```

Remove Only One

If there are multiple records and you want to delete only the first record, then set **justOne** parameter in **remove()** method.

```
>db.COLLECTION_NAME.remove(DELETION_CRITERIA,1)
```

Remove All Documents

If you don't specify deletion criteria, then MongoDB will delete whole documents from the collection. **This is equivalent of SQL's truncate command.**

```
> db.mycol.remove({})
WriteResult({ "nRemoved" : 2 })
> db.mycol.find()
>
```

MongoDB - Projection

In MongoDB, projection means selecting only the necessary data rather than selecting whole of the data of a document. If a document has 5 fields and you need to show only 3, then select only 3 fields from them.

The find() Method

MongoDB's **find()** method, explained in MongoDB Query Document accepts second optional parameter that is list of fields that you want to retrieve. In MongoDB, when

you execute **find()** method, then it displays all fields of a document. To limit this, you need to set a list of fields with value 1 or 0. 1 is used to show the field while 0 is used to hide the fields.

Syntax

The basic syntax of **find()** method with projection is as follows –

```
>db.COLLECTION_NAME.find({},{KEY:1})
```

Example

Consider the collection mycol has the following data –

```
{_id : ObjectId("507f191e810c19729de860e1"), title: "MongoDB Overview"},
{_id : ObjectId("507f191e810c19729de860e2"), title: "NoSQL Overview"},
{_id : ObjectId("507f191e810c19729de860e3"), title: "Sudaksha Overview"}
```

Following example will display the title of the document while querying the document.

```
>db.mycol.find({},{"title":1,_id:0})
{"title":"MongoDB Overview"}
{"title":"NoSQL Overview"}
{"title":"Sudaksha Overview"}
>
```

Please note **_id** field is always displayed while executing **find()** method, if you don't want this field, then you need to set it as 0.

MongoDB - Sort Records

The sort() Method

To sort documents in MongoDB, you need to use **sort()** method. The method accepts a document containing a list of fields along with their sorting order. To specify sorting order 1 and -1 are used. 1 is used for ascending order while -1 is used for descending order.

Syntax

The basic syntax of **sort()** method is as follows –

```
>db.COLLECTION_NAME.find().sort({KEY:1})
```

Example

Consider the collection myycol has the following data.

```
{_id : ObjectId("507f191e810c19729de860e1"), title: "MongoDB Overview"}
{_id : ObjectId("507f191e810c19729de860e2"), title: "NoSQL Overview"}
```

```
{_id : ObjectId("507f191e810c19729de860e3"), title: "Sudaksha Overview"}
```

Following example will display the documents sorted by title in the descending order.

```
>db.mycol.find({},{"title":1,_id:0}).sort({"title":-1})
{"title":"Sudaksha Overview"}
{"title":"NoSQL Overview"}
{"title":"MongoDB Overview"}
>
```

Please note, if you don't specify the sorting preference, then **sort()** method will display the documents in ascending order.

MongoDB - Indexing

Indexes support the efficient resolution of queries. Without indexes, MongoDB must scan every document of a collection to select those documents that match the query statement. This scan is highly inefficient and require MongoDB to process a large volume of data.

Indexes are special data structures, that store a small portion of the data set in an easy-to-traverse form. The index stores the value of a specific field or set of fields, ordered by the value of the field as specified in the index.

The createIndex() Method

To create an index, you need to use createIndex() method of MongoDB.

Syntax

The basic syntax of **createIndex()** method is as follows().

```
>db.COLLECTION_NAME.createIndex({KEY:1})
```

Here key is the name of the field on which you want to create index and 1 is for ascending order. To create index in descending order you need to use -1.

Example

```
>db.mycol.createIndex({"title":1})
{
        "createdCollectionAutomatically" : false,
        "numIndexesBefore" : 1,
        "numIndexesAfter" : 2,
        "ok" : 1
}
>
```

In **createIndex()** method you can pass multiple fields, to create index on multiple fields.

```
>db.mycol.createIndex({"title":1,"description":-1})
>
```

The dropIndex() method

You can drop a particular index using the dropIndex() method of MongoDB.

Syntax

The basic syntax of DropIndex() method is as follows().

```
>db.COLLECTION_NAME.dropIndex({KEY:1})
```

Here, "key" is the name of the file on which you want to remove an existing index. Instead of the index specification document (above syntax), you can also specify the name of the index directly as:

```
dropIndex("name_of_the_index")
```

Example

```
> db.mycol.dropIndex({"title":1})
{
        "ok" : 0,
        "errmsg" : "can't find index with key: { title: 1.0 }",
        "code" : 27,
        "codeName" : "IndexNotFound"
}
```

The dropIndexes() method

This method deletes multiple (specified) indexes on a collection.

Syntax

The basic syntax of DropIndexes() method is as follows() –

```
>db.COLLECTION_NAME.dropIndexes()
```

Example

Assume we have created 2 indexes in the named mycol collection as shown below –

```
> db.mycol.createIndex({"title":1,"description":-1})
```

Following example removes the above created indexes of mycol –

```
>db.mycol.dropIndexes({"title":1,"description":-1})
```

```
{ "nIndexesWas" : 2, "ok" : 1 }
>
```

The getIndexes() method

This method returns the description of all the indexes int the collection.

Syntax

Following is the basic syntax od the getIndexes() method –

```
db.COLLECTION_NAME.getIndexes()
```

Example

Assume we have created 2 indexes in the named mycol collection as shown below –

```
> db.mycol.createIndex({"title":1,"description":-1})
```

Following example retrieves all the indexes in the collection mycol –

```
> db.mycol.getIndexes()
[
        {
                "v" : 2,
                "key" : {
                        "_id" : 1
                },
                "name" : "_id_",
                "ns" : "test.mycol"
        },
        {
                "v" : 2,
                "key" : {
                        "title" : 1,
                        "description" : -1
                },
                "name" : "title_1_description_-1",
                "ns" : "test.mycol"
        }
]
>
```

MongoDB - Aggregation

Aggregations operations process data records and return computed results. Aggregation operations group values from multiple documents together, and can perform a variety of operations on the grouped data to return a single result. In SQL count(*) and with group by is an equivalent of MongoDB aggregation.

The aggregate() Method

For the aggregation in MongoDB, you should use **aggregate()** method.

Syntax

Basic syntax of **aggregate()** method is as follows –

```
>db.COLLECTION_NAME.aggregate(AGGREGATE_OPERATION)
```

Example

In the collection you have the following data –

```
{
  _id: ObjectId(7df78ad8902c)
  title: 'MongoDB Overview',
  description: 'MongoDB is no sql database',
  by_user: 'sudaksha',
  url: 'http://www.tutorialspoint.com',
  tags: ['mongodb', 'database', 'NoSQL'],
  likes: 100
},
{
  _id: ObjectId(7df78ad8902d)
  title: 'NoSQL Overview',
  description: 'No sql database is very fast',
  by_user: 'sudaksha',
  url: 'http://www.tutorialspoint.com',
  tags: ['mongodb', 'database', 'NoSQL'],
  likes: 10
},
{
  _id: ObjectId(7df78ad8902e)
  title: 'Neo4j Overview',
  description: 'Neo4j is no sql database',
  by_user: 'Neo4j',
  url: 'http://www.neo4j.com',
  tags: ['neo4j', 'database', 'NoSQL'],
  likes: 750
},
```

Now from the above collection, if you want to display a list stating how many tutorials are written by each user, then you will use the following **aggregate()** method –

```
> db.mycol.aggregate([{$group : {_id : "$by_user", num_tutorial : {$sum : 1}}}])
{ "_id" : "sudaksha", "num_tutorial" : 2 }
{ "_id" : "Neo4j", "num_tutorial" : 1 }
>
```

Sql equivalent query for the above use case will be **select by_user, count(*) from mycol group by by_user**.

In the above example, we have grouped documents by field **by_user** and on each occurrence of by user previous value of sum is incremented. Following is a list of available aggregation expressions.

Expression	Description	Example
$sum	Sums up the defined value from all documents in the collection.	db.mycol.aggregate([{$group : {_id : "$by_user", num_tutorial : {$sum : "$likes"}}}])
$avg	Calculates the average of all given values from all documents in the collection.	db.mycol.aggregate([{$group : {_id : "$by_user", num_tutorial : {$avg : "$likes"}}}])
$min	Gets the minimum of the corresponding values from all documents in the collection.	db.mycol.aggregate([{$group : {_id : "$by_user", num_tutorial : {$min : "$likes"}}}])
$max	Gets the maximum of the corresponding values from all documents in the collection.	db.mycol.aggregate([{$group : {_id : "$by_user", num_tutorial : {$max : "$likes"}}}])
$push	Inserts the value to an array in the resulting document.	db.mycol.aggregate([{$group : {_id : "$by_user", url : {$push: "$url"}}}])
$addToSet	Inserts the value to an array in the resulting document but does not create duplicates.	db.mycol.aggregate([{$group : {_id : "$by_user", url : {$addToSet : "$url"}}}])

$first	Gets the first document from the source documents according to the grouping. Typically this makes only sense together with some previously applied “$sort”-stage.	db.mycol.aggregate([{$group : {_id : "$by_user", first_url : {$first : "$url"}}}])
$last	Gets the last document from the source documents according to the grouping. Typically this makes only sense together with some previously applied “$sort”-stage.	db.mycol.aggregate([{$group : {_id : "$by_user", last_url : {$last : "$url"}}}])

MongoDB - Replication

Replication is the process of synchronizing data across multiple servers. Replication provides redundancy and increases data availability with multiple copies of data on different database servers. Replication protects a database from the loss of a single server. Replication also allows you to recover from hardware failure and service interruptions. With additional copies of the data, you can dedicate one to disaster recovery, reporting, or backup.

Why Replication?

- To keep your data safe
- High (24*7) availability of data
- Disaster recovery
- No downtime for maintenance (like backups, index rebuilds, compaction)
- Read scaling (extra copies to read from)
- Replica set is transparent to the application

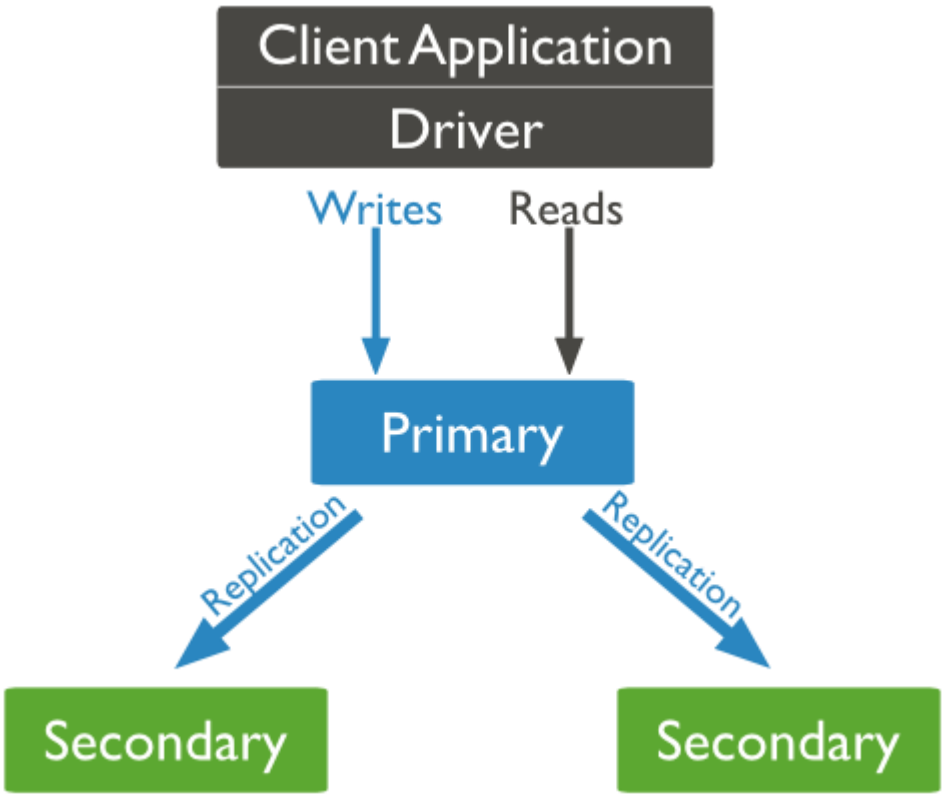
How Replication Works in MongoDB

MongoDB achieves replication by the use of replica set. A replica set is a group of **mongod** instances that host the same data set. In a replica, one node is primary

node that receives all write operations. All other instances, such as secondaries, apply operations from the primary so that they have the same data set. Replica set can have only one primary node.

- Replica set is a group of two or more nodes (generally minimum 3 nodes are required).
- In a replica set, one node is primary node and remaining nodes are secondary.
- All data replicates from primary to secondary node.
- At the time of automatic failover or maintenance, election establishes for primary and a new primary node is elected.
- After the recovery of failed node, it again join the replica set and works as a secondary node.

A typical diagram of MongoDB replication is shown in which client application always interact with the primary node and the primary node then replicates the data to the secondary nodes.



Replica Set Features

- A cluster of N nodes
- Any one node can be primary
- All write operations go to primary

- Automatic failover
- Automatic recovery
- Consensus election of primary

Set Up a Replica Set

In this tutorial, we will convert standalone MongoDB instance to a replica set. To convert to replica set, following are the steps –

- Shutdown already running MongoDB server.
-
- Start the MongoDB server by specifying -- replSet option. Following is the basic syntax of --replSet –

```
mongod --port "PORT" --dbpath "YOUR_DB_DATA_PATH" --replSet
"REPLICA_SET_INSTANCE_NAME"
```

Example

```
mongod --port 27017 --dbpath "D:\set up\mongodb\data" --replSet rs0
```

- It will start a mongod instance with the name rs0, on port 27017.
- Now start the command prompt and connect to this mongod instance.
- In Mongo client, issue the command **rs.initiate()** to initiate a new replica set.
- To check the replica set configuration, issue the command **rs.conf()**. To check the status of replica set issue the command **rs.status()**.

Add Members to Replica Set

To add members to replica set, start mongod instances on multiple machines. Now start a mongo client and issue a command **rs.add()**.

Syntax

The basic syntax of **rs.add()** command is as follows –

```
>rs.add(HOST_NAME:PORT)
```

Example

Suppose your mongod instance name is **mongod1.net** and it is running on port **27017**. To add this instance to replica set, issue the command **rs.add()** in Mongo client.

```
>rs.add("mongod1.net:27017")
>
```

You can add mongod instance to replica set only when you are connected to primary node. To check whether you are connected to primary or not, issue the command **db.isMaster()** in mongo client.

MongoDB - Sharding

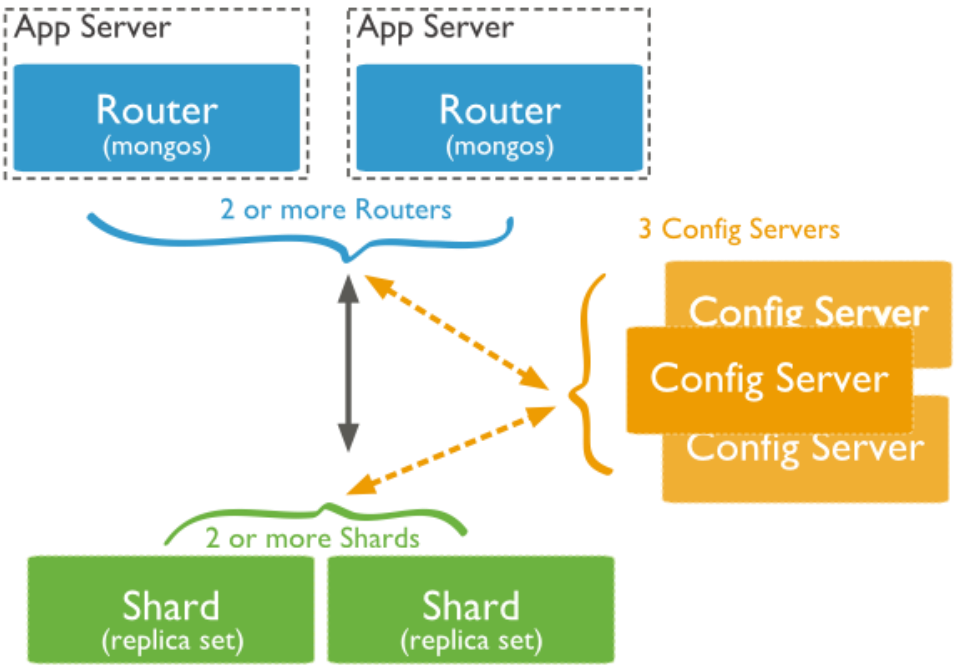
Sharding is the process of storing data records across multiple machines and it is MongoDB's approach to meeting the demands of data growth. As the size of the data increases, a single machine may not be sufficient to store the data nor provide an acceptable read and write throughput. Sharding solves the problem with horizontal scaling. With sharding, you add more machines to support data growth and the demands of read and write operations.

Why Sharding?

- In replication, all writes go to master node
- Latency sensitive queries still go to master
- Single replica set has limitation of 12 nodes
- Memory can't be large enough when active dataset is big
- Local disk is not big enough
- Vertical scaling is too expensive

Sharding in MongoDB

The following diagram shows the Sharding in MongoDB using sharded cluster.



In the following diagram, there are three main components –

- **Shards** – Shards are used to store data. They provide high availability and data consistency. In production environment, each shard is a separate replica set.
- **Config Servers** – Config servers store the cluster's metadata. This data contains a mapping of the cluster's data set to the shards. The query router uses this metadata to target operations to specific shards. In production environment, sharded clusters have exactly 3 config servers.
- **Query Routers** – Query routers are basically mongo instances, interface with client applications and direct operations to the appropriate shard. The query router processes and targets the operations to shards and then returns results to the clients. A sharded cluster can contain more than one query router to divide the client request load. A client sends requests to one query router. Generally, a sharded cluster have many query routers.