MongoDB introduction

21 March 2022 18:36

- MongoDB is a No SQL database. It is an open-source, cross-platform, document-oriented database written in C++.
- MongoDB is an open-source document database that provides high performance, high availability, and automatic scaling
- MongoDB was developed by a NewYork based organization named 10gen which is now known as MongoDB Inc
- All the modern applications require big data, fast features development, flexible deployment, and the older database systems not competent enough, so the MongoDB was needed.

The primary purpose of building MongoDB is:

  - Scalability
  - Performance
  - High Availability
  - Scaling from single server deployments to large, complex multi-site architectures.
  - Key points of MongoDB
  - Develop Faster
  - Deploy Easier
  - Scale Bigger
- MongoDB stores data as documents, so it is known as document-oriented database.

Features of MongoDB

These are some important features of MongoDB:

1. Support ad hoc queries

In MongoDB, you can search by field, range query and it also supports regular expression searches.

2. Indexing

You can index any field in a document.

3. Replication

MongoDB supports Master Slave replication.

A master can perform Reads and Writes and a Slave copies data from the master and can only be used for reads or back up (not writes)

4. Duplication of data

MongoDB can run over multiple servers. The data is duplicated to keep the system up and also keep its running condition in case of hardware failure.

5. Load balancing

It has an automatic load balancing configuration because of data placed in shards.

6. Supports map reduce and aggregation tools.

7. Uses JavaScript instead of Procedures.

8. It is a schema-less database written in C++.

9. Provides high performance.

10. Stores files of any size easily without complicating your stack.

11. Easy to administer in the case of failures.

12. It also supports:

JSON data model with dynamic schemas

Auto-sharding for horizontal scalability

Built in replication for high availability

Databases can be divided in 3 types:

1. RDBMS (Relational Database Management System)
2. OLAP (Online Analytical Processing)
3. NoSQL (recently developed database)

- NoSQL mean non relational databases
- As Relational Databases could not handle the big data, NoSQL database have been developed.

Advantages of NoSQL

- It supports query language.
- It provides fast performance.
- It provides horizontal scalability.

- In recent days, MongoDB is a new and popularly used database. It is a document based, non relational database provider.

MongoDB Advantages

- **MongoDB is schema less**. It is a document database in which one collection holds different documents.
- There may be **difference between number of fields, content and size of the document** from one to other.
- **Structure of a single object is clear** in MongoDB.
- There are **no complex joins** in MongoDB.
- MongoDB provides the **facility of deep query** because it supports a powerful dynamic query on documents.
- It is very **easy to scale**.
- It **uses internal memory for storing working sets** and this is the reason of its fast access.

Performance analysis of MongoDB and RDBMS

- In relational database (RDBMS) tables are using as storing elements, while in MongoDB collection is used.
- In the RDBMS, we have multiple schema and in each schema we create tables to store data while, MongoDB is a document oriented database in which data is written in BSON format which is a JSON like format.
- MongoDB is almost 100 times faster than traditional database systems.

MongoDB datatypes

Data Types	Description
String	String is the most commonly used datatype. It is used to store data. A string must be UTF 8 valid in mongodb.
Integer	Integer is used to store the numeric value. It can be 32 bit or 64 bit depending on the server you are using.
Boolean	This datatype is used to store boolean values. It just shows YES/NO values.
Double	Double datatype stores floating point values.
Min/Max Keys	This datatype compare a value against the lowest and highest bson elements.
Arrays	This datatype is used to store a list or multiple values into a single key.
Object	Object datatype is used for embedded documents.
Null	It is used to store null values.
Symbol	It is generally used for languages that use a specific type.
Date	This datatype stores the current date or time in unix time format. It makes you possible to specify your own date time by creating object of date and pass the value of date, month, year into it.

MongoDB Create and Drop Database, Collections

22 March 2022 16:39

- If the database is not exists then it creates the new database or else it returns already created database. Use the following command to create a new database.

```
>use javatpointdb
```

```
Swithched to db javatpointdb
```

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

```
>db
```

```
javatpointdb
```

- Here if we want to check all the created databases we need to use

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

```
>db
```

```
javatpointdb
```

- The database is visible only if it has at least one document.
- Command to insert a document.

```
>db.movie.insert({"name":"javatpoint"})
```

```
WriteResult({ "nInserted": 1})
```

If you want to **delete the database "javatpointdb"**, use the dropDatabase() command as follows:

```
>use javatpointdb
```

```
switched to the db javatpointdb
```

```
>db.dropDatabase()
```

```
{ "dropped": "javatpointdb", "ok": 1}
```

MongoDB create collection

- Collection is similar to tables, below we are creating a collection.

Let's take an **example to create collection**. In this example, we are going to create a collection name SSSIT.

```
>use test
```

```
switched to db test
```

```
>db.createCollection("SSSIT")
```

```
{ "ok" : 1 }
```

To **check the created collection**, use the command "show collections".

```
>show collections
```

```
SSSIT
```

- Even if we not create a collection and insert any document by giving the name of collection which is not created leads to create a new collection automatically.
- We can find the contents of the collection using find() method.

```
>db.SSSIT.insert({"name" : "seomount"})
>show collections
SSSIT
```

If you want to see the inserted document, use the find() command.

Syntax:

```
db.collection_name.find()
```

MongoDB Drop collection

Now **drop the collection** with the name SSSIT:

```
>db.SSSIT.drop()
```

```
True
```

MongoDB Shell

22 March 2022 22:08

- The shell is a full-featured JavaScript interpreter. It is capable of running Arbitrary JavaScript

Let us take a simple mathematical program:

```
>x= 100
100
>x/ 5;
20
```

You can also use the JavaScript libraries

```
> "Hello, World!".replace("World", "MongoDB");
```

```
Hello, MongoDB!
```

You can even define and call JavaScript functions

```
> function factorial (n) {
... if (n <= 1) return 1;
... return n * factorial(n - 1);
... }
> factorial (5);
120
```

MongoDB Documents

22 March 2022 17:03

- In MongoDB, the db.collection.insert() method is used to add or insert new documents into a collection in your database.
- Upsert is an operation that performs either an update of existing document or an insert of new document if the document to modify does not exist.

```
db.javatpoint.insert(
  {
   course: "java",
   details: {
     duration: "6 months",
     Trainer: "Sonoo jaiswal"
   },
   Batch: [ { size: "Small", qty: 15 }, { size: "Medium", qty: 25 } ],
   category: "Programming language"
  }
)
```

After the successful insertion of the document, the operation will return a WriteResult object with its status.

Output:

```
WriteResult({ "nInserted" : 1 })
```

Here the **nInserted** field specifies the number of documents inserted. If an error is occurred then the **WriteResult** will specify the error information.

Check the inserted documents

If the insertion is successful, you can view the inserted document by the following query.

```
>db.javatpoint.find()
```

- If you want to insert multiple documents in a collection, you have to pass an array of documents to the db.collection.insert() method.

Create an array of documents

Define a variable named Allcourses that hold an array of documents to insert.

```
var Allcourses =
  [
   {
    Course: "Java",
    details: { Duration: "6 months", Trainer: "Sonoo Jaiswal" },
    Batch: [ { size: "Medium", qty: 25 } ],
    category: "Programming Language"
   },
   {
    Course: ".Net",
    details: { Duration: "6 months", Trainer: "Prashant Verma" },
    Batch: [ { size: "Small", qty: 5 }, { size: "Medium", qty: 10 }, ],
    category: "Programming Language"
   },
```

```
    category: "Programming Language"
  },
  {
    Course: "Web Designing",
    details: { Duration: "3 months", Trainer: "Rashmi Desai" },
    Batch: [ { size: "Small", qty: 5 }, { size: "Large", qty: 10 } ],
    category: "Programming Language"
  }
];
```

Inserts the documents

Pass this Allcourses array to the db.collection.insert() method to perform a bulk insert.

```
> db.javatpoint.insert( Allcourses );
```

Mongo DB Update Documents

- update() method is used to update or modify the existing documents of a collection.
- Suppose if we think that we have inserted the following document inside the collection

```
db.javatpoint.insert(
  {
    course: "java",
    details: {
      duration: "6 months",
      Trainer: "Sonoo jaiswal"
    },
    Batch: [ { size: "Small", qty: 15 }, { size: "Medium", qty: 25 } ],
    category: "Programming language"
  }
)
```

- Now we need to update the following document so we use the following query to update
- Here we use key as the java and update the course name of that.

Update the existing course "java" into "android":

```
>db.javatpoint.update({'course':'java'},{$set:{'course':'android'}})
```

Check the updated document in the collection:

```
>db.javatpoint.find()
```

Output:

```
{ "_id" : ObjectId("56482d3e27e53d2dbc93cef8"), "course" : "android", "details" :
{ "duration" : "6 months", "Trainer" : "Sonoo jaiswal" }, "Batch" :
[ {"size" : "Small", "qty" : 15 }, { "size" : "Medium", "qty" : 25 } ],
 "category" : "Programming language" }
```

- We are using set operator here to set the value of a set.

MongoDB Remove Documents

- **Remove all documents**

```
db.javatpoint.remove({})
```

- **Remove all documents that match a condition (any number of documents will be removed)**

```
db.javatpoint.remove( { type : "programming language" } )
```

- **Remove a single document that match a condition (the first document that matches the condition)**

```
db.javatpoint.remove( { type : "programming language" }, 1 )
```

MongoDB Query Documents

- To check all the documents in a collection we can use find() function for the document name.

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

SQL vs MongoDB

SQL Terms	MongoDB Terms
database	Database
table	Collection
row	document or BSON document
column	field
index	index
table joins	$lookup, embedded document
primary key	primary key
In SQL, we can specify any unique column or column combination as the primary key.	In MongoDB, we don't need to set the primary key. The _id field is automatically set to the primary key.
aggregation	aggregation pipeline
SELECT INTO NEW_TABLE	$out
MERGE INTO TABLE	$merge
transactions	transactions

MongoDB main Queries

22 March 2022 18:03

Create and Alter commands

SQL statements	MongoDB statements
CREATE TABLE JavaTpoint (id MEDIUMINT NOT NULL AUTO_INCREMENT, user_id Varchar(20), age Number, status char(1), PRIMARY KEY (id))	db.createCollection (" JavaTpoint ")
ALTER TABLE JavaTpoint ADD join_date DATETIME	db.JavaTpoint.updateMany({ }, { $set: { join_date: new Date() } })
ALTER TABLE JavaTpoint DROP COLUMN join_date	db.JavaTpoint.updateMany({ }, { $unset: { "join_date": "" } })
CREATE INDEX idx_user_id_asc ON JavaTpoint (user_id)	db.people.createIndex ({ user_id: 1 })
CREATE INDEX idx_user_id_asc ON people (user_id)	db.people.createIndex({ user_id: 123, age: 1})
DROP TABLE people	db.people.drop ()

MongoDB and SQL Insert Statement

SQL Insert statement	MongoDB insert statement
INSERT INTO JavaTpoint (user_id, age, status) VALUES ("mongo", 45, "A")	db.JavaTpoint.insertOne({ user_id: "mongo", age: 18, status: "A" })

SQL and Mongo DB Select Command

SQL SELECT Statement	MongoDB find() Statement
SELECT * FROM JavaTpoint	db.JavaTpoint.find()
SELECT id, user_id, status FROM JavaTpoint	db.JavaTpoint.find({ }, { user_id: 1, status: 1 })
SELECT user_id, status FROM JavaTpoint	db.JavaTpoint.find({ }, { user_id: 1, status: 1, _id: 0 })
SELECT * FROM JavaTpoint WHERE status = "B"	db.JavaTpoint.find({ status: "A" })
SELECT user_id, status FROM JavaTpoint WHERE status = "A"	db.javaTpoint.find({ status: "A" }, { user_id: 1, status: 1, _id: 0 }
SELECT * FROM JavaTpoint WHERE status != "A"	db.JavaTpoint.find({ status: { $ne: "A" } })
SELECT * FROM JavaTpoint WHERE status = "A" AND age = 50	db.JavaTpoint.find({ status: "A", age: 50 })
SELECT * FROM JavaTpoint WHERE status = "A" OR age = 50	db.JavaTpoint.find({ $or: [{ status: "A" } , { age: 50 }] })
SELECT * FROM JavaTpoint WHERE age > 25	db.JavaTpoint.find({ age: { $gt: 25 } })
SELECT * FROM JavaTpoint WHERE age < 25	Db.JavaTpoint.find({ age: { $lt: 25 } })
SELECT * FROM JavaTpoint WHERE age > 25 AND age <= 50	db.JavaTpoint.find({ age: { $gt: 25, $lte: 50 } })
SELECT * FROM JavaTpoint WHERE user_id like "%bc%"	db.JavaTpoint.find({ user_id: /bc/ }) -or- db.JavaTpoint.find({ user_id: { $regex: /bc/ } })
SELECT *	db.JavaTpoint.find({ user_id: /^bc/ })

SELECT * FROM JavaTpoint WHERE user_id like "bc%"	db.JavaTpoint.find({ user_id: /^bc/ }) -or- db.JavaTpoint.find({ user_id: { $regex: /^bc/ } })
SELECT * FROM JavaTPoint WHERE status = "A" ORDER BY user_id ASC	db. JavaTPoint. find({ status: "A" }). sort({ user_id: 1 })
SELECT * FROM JavaTPoint WHERE status = "A" ORDER BY user_id ASC	db. JavaTPoint. find({ status: "A" }). sort({ user_id: 1 })
SELECT * FROM JavaTPoint WHERE status = "A" ORDER BY user_id DESC	db. JavaTPoint. find({ status: "A" }). sort({ user_id: -1 })
SELECT * FROM JavaTPoint WHERE status = "A" ORDER BY user_id DESC	db. JavaTPoint. find({ status: "A" }). sort({ user_id: -1 })
SELECT COUNT(*) FROM JavaTPoint	db. JavaTPoint. count() or db. JavaTPoint. find(). count()
SELECT COUNT(user_id) FROM JavaTPoint	db. JavaTPoint.count({ user_id: { $exists: true } }) or db. JavaTPoint.find({ user_id: { $exists: true } }).count()
SELECT COUNT(*) FROM JavaTPoint WHERE age > 30	db. JavaTPoint.count({ age: { $gt: 30 } }) or db. JavaTPoint.find({ age: { $gt: 30 } }).count()
SELECT DISTINCT(status) FROM JavaTPoint	db. JavaTPoint.aggregate([{ $group : { _id : "$status" } }]) or, for distinct value sets that do not exceed the BSON size limit db. JavaTPoint.distinct("status")

SELECT * FROM JavaTPoint LIMIT 1	db. JavaTPoint.findOne() or db. JavaTPoint.find(). limit(1)
SELECT * FROM JavaTPoint LIMIT 5 SKIP 10	db. JavaTPoint.find(). limit(5). skip(10)
EXPLAIN SELECT * FROM JavaTPoint WHERE status = "A"	db. JavaTPoint. find({ status: "A" }). explain()

SQL and MongoDB Update Statements

SQL Update Statements	MongoDB updateMany() Statements
UPDATE JavaTpoint SET status = "C" WHERE age > 25	db.JavaTpoint.updateMany({ age: { $gt: 25 } }, { $set: { status: "C" } })
UPDATE JavaTpoint SET age = age + 3 WHERE status = "A"	db.JavaTpoint.updateMany({ status: "A" } , { $inc: { age: 3 } })

MongoDB Delete Statements

SQL Delete Statements	MongoDB deleteMany() Statements
DELETE FROM JavaTpoint WHERE status = "D"	db.JavaTpoint.deleteMany({ status: "D" })
DELETE FROM JavaTpoint	db.JavaTpoint.deleteMany({ })

MongoDB SearchText

22 March 2022 18:38

- Suppose if we have such document

```
> db.content.find().pretty()
{
        "_id" : ObjectId("603622eef19652db63812eb5"),
        "name" : "Rohit",
        "line" : "I love dogs and cats"
}
{
        "_id" : ObjectId("603622eef19652db63812eb6"),
        "name" : "Priya",
        "line" : "I love dogs and cats"
}
{
        "_id" : ObjectId("603622eef19652db63812eb7"),
        "name" : "Suman",
        "line" : "I dont like dogs and cats but i like cow"
}
>
```

- Now we need to search any word or phrase on the document then first we need to tell where to search first.

```
> db.content.createIndex({name:"text",line:"text"})
{
        "createdCollectionAutomatically" : false,
        "numIndexesBefore" : 1,
        "numIndexesAfter" : 2,
        "ok" : 1
}
```

- Here we are telling to search the text that we give under name and line fields of the content document that we have created.
- Now we need to give the word to search under these.

```
> db.content.find({$text:{$search:"love"}})
{ "_id" : ObjectId("603622eef19652db63812eb6"), "name" : "Priya", "line" : "I lo
ve dogs and cats" }
{ "_id" : ObjectId("603622eef19652db63812eb5"), "name" : "Rohit", "line" : "I lo
ve dogs and cats" }
```

```
> db.content.find({$text:{$search:"dog"}})
{ "_id" : ObjectId("603622eef19652db63812eb6"), "name" : "Priya", "line" : "I lo
ve dogs and cats" }
{ "_id" : ObjectId("603622eef19652db63812eb5"), "name" : "Rohit", "line" : "I lo
ve dogs and cats" }
{ "_id" : ObjectId("603622eef19652db63812eb7"), "name" : "Suman", "line" : "I do
nt like dogs and cats but i like cow" }
```

- For searching of words we use the following method. But while we try to find a phrase we can not write it like the word because , in general using above syntax a phrase is interpreted as group of individual words. So it does not find phrases it just find the words.

Wrong method to search a phrase (Right method to search multiple words)

- **Document we have**

```
> db.content.find().pretty()
{
        "_id" : ObjectId("603622eef19652db63812eb5"),
        "name" : "Rohit",
        "line" : "I love dogs and cats"
}
{
        "_id" : ObjectId("603622eef19652db63812eb6"),
        "name" : "Priya",
        "line" : "I love dogs and cats"
}
{
        "_id" : ObjectId("603622eef19652db63812eb7"),
        "name" : "Suman",
        "line" : "I dont like dogs and cats but i like cow"
}
```

- **This is not correct query to find a phrase , it is a way to search multiple words at a time.**

```
> db.content.find({$text:{$search:"I love dogs"}})
{ "_id" : ObjectId("603622eef19652db63812eb6"), "name" : "Priya", "line" : "I lo
ve dogs and cats" }
{ "_id" : ObjectId("603622eef19652db63812eb5"), "name" : "Rohit", "line" : "I lo
ve dogs and cats" }
{ "_id" : ObjectId("603622eef19652db63812eb7"), "name" : "Suman", "line" : "I do
nt like dogs and cats but i like cow" }
```

- **The correct method**

```
> db.content.find({$text:{$search:"\"I love dogs\""}})
{ "_id" : ObjectId("603622eef19652db63812eb6"), "name" : "Priya", "line" : "I lo
ve dogs and cats" }
{ "_id" : ObjectId("603622eef19652db63812eb5"), "name" : "Rohit", "line" : "I lo
ve dogs and cats" }
```

- **Suppose we need to find a term in documents without a word in it.**

```
> db.content.find({$text:{$search:"dog -cow"}})
{ "_id" : ObjectId("603622eef19652db63812eb6"), "name" : "Priya", "line" : "I lo
ve dogs and cats" }
{ "_id" : ObjectId("603622eef19652db63812eb5"), "name" : "Rohit", "line" : "I lo
ve dogs and cats" }
```

- **It is saying that the search document must have dog but not cat.**
- **If we want to sort the documents by doing search and based on the meta score**

```
db.library.find(
  { $text: { $search: "java" } },
  { score: { $meta: "textScore" } }
).sort( { score: { $meta: "textScore" } } )
```

In the above example we explicitly project the meta textScore field to sort the result in order of relevance score.

MongoDB Shell Collection Methods

Wednesday, March 23, 2022 4:26 PM

#1: db.collection.aggregate(pipeline, option)

The aggregate method calculates mass values for the data in a collection/table or in a view.

Pipeline: It is an array of mass data operations or stages. It can accept the pipeline as a separate argument, not as an element in an array. If the pipeline is not specified as an array, then the second parameter will not be specified.

Option: A document that passes the aggregate command. It will be available only when you specify the pipeline as an array.

We use the following document

```
{ _id: 1, book_id: "Java", ord_date: ISODate("2012-11-02T17:04:11.102Z"), status: "A", amount: 50 }
{ _id: 0, book_id: "MongoDB", ord_date: ISODate("2013-10-01T17:04:11.102Z"), status: "A", amount: 100 }
{ _id: 0.01, book_id: "DBMS", ord_date: ISODate("2013-10-12T17:04:11.102Z"), status: "D", amount: 25 }
{ _id: 2, book_id: "Python", ord_date: ISODate("2013-10-11T17:04:11.102Z"), status: "D", amount: 125 }
{ _id: 0.02, book_id: "SQL", ord_date: ISODate("2013-11-12T17:04:11.102Z"), status: "A", amount: 25 }
```

Complete Mongo

Wednesday, March 23, 2022 7:29 PM

CREATE A DATABASE

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.

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":"tutorials point"})
>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.

j

DROP DATABASE

The dropDatabase() Method

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

Example

First, check the list of available databases by using the command, **show dbs**.

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

If you want to delete new database **<mydb>**, then **dropDatabase()** command would be as follows −

```
>use mydb
switched to db mydb
>db.dropDatabase()
>{ "dropped" : "mydb", "ok" : 1 }
>
```

Now check list of databases.

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

CREATE A COLLECTION

The createCollection() Method

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

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.

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, ma
"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
>
```

DROP A COLLECTION

The drop() Method

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

Example

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

```
>use mydb
switched to db mydb
>show collections
mycol
mycollection
system.indexes
tutorialspoint
>
```

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

```
>db.mycollection.drop()
true
>
```

Again check the list of collections into database.

```
>show collections
mycol
system.indexes
tutorialspoint
>
```

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

MONGO DB 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.

INSERT A DOCUMENT

The insert() Method

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

Example

```
> db.users.insert({
... _id : ObjectId("507f191e810c19729de860ea"),
... title: "MongoDB Overview",
... description: "MongoDB is no sql database",
... by: "tutorials point",
... 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
```

INSERTING MULTIPLE DOCUMENTS AT THE SAME TIME USING ARRAY

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: "tutorials point",
                url: "http://www.tutorialspoint.com",
                tags: ["mongodb", "database", "NoSQL"],
                likes: 100
        },
        {
        title: "NoSQL Database",
        description: "NoSQL database doesn't have tables",
        by: "tutorials point",
        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
                }
        ]
}
])
```

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: "Radhika",
                Last_Name: "Sharma",
                Date_Of_Birth: "1995-09-26",
                e_mail: "radhika_sharma.123@gmail.com",
                phone: "9848022338"
        })
{
        "acknowledged" : true,
        "insertedId" : ObjectId("5dd62b4070fb13eec3963bea")
}
```

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

```
> db.empDetails.insertMany(
        [
                {
                        First_Name: "Radhika",
                        Last_Name: "Sharma",
                        Date_Of_Birth: "1995-09-26",
                        e_mail: "radhika_sharma.123@gmail.com",
                        phone: "9000012345"
                },
                {
                        First_Name: "Rachel",
                        Last_Name: "Christopher",
                        Date_Of_Birth: "1990-02-16",
                        e_mail: "Rachel_Christopher.123@gmail.com",
                        phone: "9000054321"
                },
                {
                        First_Name: "Fathima",
                        Last_Name: "Sheik",
                        Date_Of_Birth: "1990-02-16",
                        e_mail: "Fathima_Sheik.123@gmail.com",
                        phone: "9000054321"
                }
        ]
)
```

Find() method

The find() Method

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

- 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: "tutorials point",
                url: "http://www.tutorialspoint.com",
                tags: ["mongodb", "database", "NoSQL"],
                likes: 100
        },
        {
                title: "NoSQL Database",
                description: "NoSQL database doesn't have tables",
                by: "tutorials point",
                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"
{ "_id" : ObjectId("5dd4e2cc0821d3b44607534d"), "title" : "NoSQL Database", "description" :
>
```

```
> db.mycol.find().pretty()
{
        "_id" : ObjectId("5dd4e2cc0821d3b44607534c"),
        "title" : "MongoDB Overview",
        "description" : "MongoDB is no SQL database",
        "by" : "tutorials point",
        "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" : "tutorials point",
        "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
                }
        ]
}
```

```
> db.mycol.findOne({title: "MongoDB Overview"})
{
        "_id" : ObjectId("5dd6542170fb13eec3963bf0"),
        "title" : "MongoDB Overview",
        "description" : "MongoDB is no SQL database",
        "by" : "tutorials point",
        "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>:{$eg; <value>}}	db.mycol.find({"by":"tutorials point"}).pretty()	where by = 'tutorials point'
Less Than	{<key>:{$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": {$ne:50}}).pretty()	where likes I= 50
Values in an array	{<key>:{$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 'tutorials point' and whose title is 'MongoDB Overview'.

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

For the above given example, equivalent where clause will be **' where by = 'tutorials point' 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 'tutorials point' or whose title is 'MongoDB Overview'.

```
>db.mycol.find({$or:[{"by":"tutorials point"},{"title": "MongoDB Overview"}]}).pretty(
{
   "_id": ObjectId(7df78ad8902c),
   "title": "MongoDB Overview",
   "description": "MongoDB is no sql database",
   "by": "tutorials point",
   "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 'tutorials point'. Equivalent SQL where clause is **'where likes>10 AND (by = 'tutorials point' OR title = 'MongoDB Overview')'**

```
>db.mycol.find({"likes": {$gt:10}, $or: [{"by": "tutorials point"},
   {"title": "MongoDB Overview"}]}).pretty()
{
   "_id": ObjectId(7df78ad8902c),
   "title": "MongoDB Overview",
   "description": "MongoDB is no sql database",
   "by": "tutorials point",
   "url": "http://www.tutorialspoint.com",
   "tags": ["mongodb", "database", "NoSQL"],
   "likes": "100"
}
>
```

NOR IN MONGODB

Example

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

```
db.empDetails.insertMany(
        [
                {
                        First_Name: "Radhika",
                        Last_Name: "Sharma",
                        Age: "26",
                        e_mail: "radhika_sharma.123@gmail.com",
                        phone: "9000012345"
                },
                {
                        First_Name: "Rachel",
                        Last_Name: "Christopher",
                        Age: "27",
                        e_mail: "Rachel_Christopher.123@gmail.com",
                        phone: "9000054321"
                },
                {
                        First_Name: "Fathima",
                        Last_Name: "Sheik",
                        Age: "24",
                        e_mail: "Fathima_Sheik.123@gmail.com",
                        phone: "9000054321"
                }
        ]
)
```

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

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

NOT IN MONGODB

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" : "Fathima",
        "Last_Name" : "Sheik",
        "Age" : "24",
        "e_mail" : "Fathima_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.

Updating single document

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":"Tutorials Point 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":"Tutorials Point Overview"}
>
```

Updating multiple documents

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.

Example

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

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

MongoDB findOneAndUpdate() method

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

Example

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

```
> db.empDetails.insertMany(
       [
              {
                     First_Name: "Radhika",
                     Last_Name: "Sharma",
                     Age: "26",
                     e_mail: "radhika_sharma.123@gmail.com",
                     phone: "9000012345"
              },
              {
                     First_Name: "Rachel",
                     Last_Name: "Christopher",
                     Age: "27",
                     e_mail: "Rachel_Christopher.123@gmail.com",
                     phone: "9000054321"
              },
              {
                     First_Name: "Fathima",
                     Last_Name: "Sheik",
                     Age: "24",
                     e_mail: "Fathima_Sheik.123@gmail.com",
                     phone: "9000054321"
              }
       ]
)
```

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

```
> db.empDetails.findOneAndUpdate(
       {First_Name: 'Radhika'},
       { $set: { Age: '30',e_mail: 'radhika_newemail@gmail.com'}}
)
{
       "_id" : ObjectId("5dd6636870fb13eec3963bf5"),
       "First_Name" : "Radhika",
       "Last_Name" : "Sharma",
       "Age" : "30",
       "e_mail" : "radhika_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: 'Radhika'},
       { $set: { Age: '30',e_mail: 'radhika_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 −

```
.find()
:Id("5dd6636870fb13eec3963bf5"), "First_Name" : "Radhika", "Last_Name" : "Sharma", "Age" : "00", "e_mail"
:Id("5dd6636870fb13eec3963bf6"), "First_Name" : "Rachel", "Last_Name" : "Christopher", "Age" : "00", "e_ma
:Id("5dd6636870fb13eec3963bf7"), "First_Name" : "Fathima", "Last_Name" : "Sheik", "Age" : "24", "e_mail" :
```

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.

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: "Tutorials Point 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" : "Tutorials Point 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.

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: "Tutorials Point 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":"Tutorials Point 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 LIMIT RECORDS

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: "Tutorials Point Overview"}
```

Following example will display only two documents while querying the document.

```
>db.mycol.find({},{"title":1,_id:0}).limit(2)
{"title":"MongoDB Overview"}
{"title":"NoSQL Overview"}
>
```

If you don't specify the number argument in **limit()** method then it will display all documents from the collection.

MONGODB SKIP METHODS

MongoDB Skip() Method

Apart from limit() method, there is one more method **skip()** which also accepts number type argument and is used to skip the number of documents.

Example

Following example will display only the second document.

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

Please note, the default value in **skip()** method is 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.

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: "Tutorials Point 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":"Tutorials Point 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

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})
>
```

Example

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

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 }
```

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 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.

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.

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()**.

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.

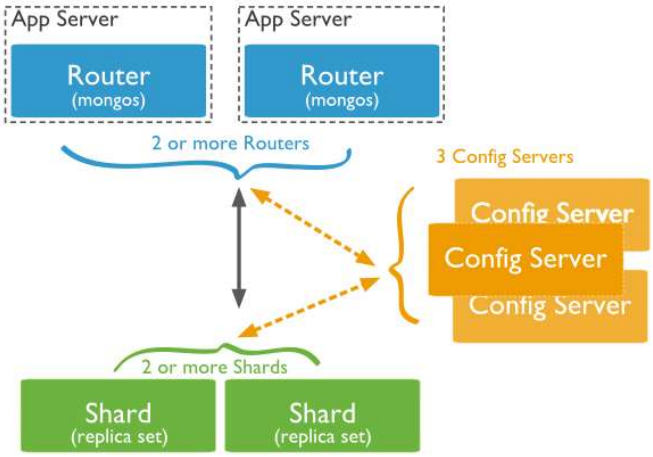
MONGODB SHARDING

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.

- 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.

MONGODB DUMP DATA

- To create backup of database in MongoDB, you should use mongodump command. This command will dump the entire data of your server into the dump directory.

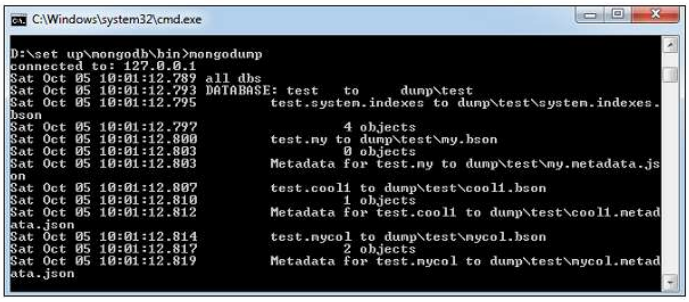
Example

Start your mongod server. Assuming that your mongod server is running on the localhost and port 27017, open a command prompt and go to the bin directory of your mongodb instance and type the command **mongodump**

Consider the mycol collection has the following data.

```
>mongodump
```

The command will connect to the server running at **127.0.0.1** and port **27017** and back all data of the server to directory **/bin/dump/**. Following is the output of the command −



mongodump --dbpath DB_PATH --out BACKUP_DIRECTORY	This command will backup only specified database at specified path.	mongodump --dbpath /data/db/ --out /data/backup/
mongodump --collection COLLECTION --db DB_NAME	This command will backup only specified collection of specified database.	mongodump --collection mycol --db test

MONGODB JAVA

Installation

Before you start using MongoDB in your Java programs, you need to make sure that you have MongoDB CLIENT and Java set up on the machine. You can check Java tutorial for Java installation on your machine. Now, let us check how to set up MongoDB CLIENT.

- You need to download the jar **mongodb-driver-3.11.2.jar and its dependency mongodb-driver-core-3.11.2.jar.**. Make sure to download the latest release of these jar files.
- You need to include the downloaded jar files into your classpath.

Connect to Database

To connect database, you need to specify the database name, if the database doesn't exist then MongoDB creates it automatically.

Following is the code snippet to connect to the database −

```
import com.mongodb.client.MongoDatabase;
import com.mongodb.MongoClient;
import com.mongodb.MongoCredential;
public class ConnectToDB {

   public static void main( String args[] ) {

      // Creating a Mongo client
      MongoClient mongo = new MongoClient( "localhost" , 27017 );

      // Creating Credentials
      MongoCredential credential;
      credential = MongoCredential.createCredential("sampleUser", "myDb
         "password".toCharArray());
      System.out.println("Connected to the database successfully");

      // Accessing the database
      MongoDatabase database = mongo.getDatabase("myDb");
      System.out.println("Credentials ::"+ credential);
   }
}
```

Now, let's compile and run the above program to create our database myDb as shown below.

```
$javac ConnectToDB.java
$java ConnectToDB
```

On executing, the above program gives you the following output.

```
Connected to the database successfully
Credentials ::MongoCredential{
   mechanism = null,
   userName = 'sampleUser',
   source = 'myDb',
   password = <hidden>,
   mechanismProperties = {}
}
```

Create a Collection

To create a collection, **createCollection()** method of **com.mongodb.client.MongoDatabase** class is used.

Following is the code snippet to create a collection −

```
import com.mongodb.client.MongoDatabase;
import com.mongodb.MongoClient;
import com.mongodb.MongoCredential;
public class CreatingCollection {

   public static void main( String args[] ) {

      // Creating a Mongo client
      MongoClient mongo = new MongoClient( "localhost" , 27017 );

      // Creating Credentials
      MongoCredential credential;
      credential = MongoCredential.createCredential("sampleUser", "myDb",
         "password".toCharArray());
      System.out.println("Connected to the database successfully");

      //Accessing the database
      MongoDatabase database = mongo.getDatabase("myDb");

      //Creating a collection
      database.createCollection("sampleCollection");
      System.out.println("Collection created successfully");
   }
}
```

On compiling, the above program gives you the following result −

```
Connected to the database successfully
Collection created successfully
```

Getting/Selecting a Collection

To get/select a collection from the database, **getCollection()** method of **com.mongodb.client.MongoDatabase** class is used.

Following is the program to get/select a collection −

```
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import org.bson.Document;
import com.mongodb.MongoClient;
import com.mongodb.MongoCredential;
public class selectingCollection {

   public static void main( String args[] ) {

      // Creating a Mongo client
      MongoClient mongo = new MongoClient( "localhost" , 27017 );

      // Creating Credentials
      MongoCredential credential;
      credential = MongoCredential.createCredential("sampleUser", "myDb
         "password".toCharArray());
      System.out.println("Connected to the database successfully");

      // Accessing the database
      MongoDatabase database = mongo.getDatabase("myDb");

      // Creating a collection
      System.out.println("Collection created successfully");
      // Retrieving a collection
      MongoCollection<Document> collection = database.getCollection("my
      System.out.println("Collection myCollection selected successfully
   }
}
```

On compiling, the above program gives you the following result −

```
Connected to the database successfully
Collection created successfully
Collection myCollection selected successfully
```

Insert a Document

To insert a document into MongoDB, **insert()** method of **com.mongodb.client.MongoCollection** class is used.

Following is the code snippet to insert a document −

```
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import org.bson.Document;
import com.mongodb.MongoClient;
public class InsertingDocument {
        public static void main( String args[] ) {

        // Creating a Mongo client
        MongoClient mongo = new MongoClient( "localhost" , 27017 );

        // Accessing the database
        MongoDatabase database = mongo.getDatabase("myDb");

        // Creating a collection
        database.createCollection("sampleCollection");
        System.out.println("Collection created successfully");

        // Retrieving a collection
        MongoCollection<Document> collection = database.getCollection("
        System.out.println("Collection sampleCollection selected succes
        Document document = new Document("title", "MongoDB")
        .append("description", "database")
        .append("likes", 100)
        .append("url", "http://www.tutorialspoint.com/mongodb/")
        .append("by", "tutorials point");

        //Inserting document into the collection
        collection.insertOne(document);
        System.out.println("Document inserted successfully");
}
```

On compiling, the above program gives you the following result −

```
Connected to the database successfully
Collection sampleCollection selected successfully
Document inserted successfully
```

Retrieve All Documents

To select all documents from the collection, **find()** method of **com.mongodb.client.MongoCollection** class is used. This method returns a cursor, so you need to iterate this cursor.

Following is the program to select all documents −

```
import com.mongodb.client.FindIterable;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import org.bson.Document;
import com.mongodb.MongoClient;
import com.mongodb.MongoCredential;
public class RetrievingAllDocuments {
        public static void main( String args[] ) {

                // Creating a Mongo client
                MongoClient mongo = new MongoClient( "localhost" , 27017 );

                // Creating Credentials
                MongoCredential credential;
                credential = MongoCredential.createCredential("sampleUser", "myDb", "password".toCharArra
                System.out.println("Connected to the database successfully");

                // Accessing the database
                MongoDatabase database = mongo.getDatabase("myDb");
```

```
            // Retrieving a collection
            MongoCollection<Document> collection = database.getCollection("sampleCollection");
            System.out.println("Collection sampleCollection selected successfully");
            Document document1 = new Document("title", "MongoDB")
            .append("description", "database")
            .append("likes", 100)
            .append("url", "http://www.tutorialspoint.com/mongodb/")
            .append("by", "tutorials point");
            Document document2 = new Document("title", "RethinkDB")
            .append("description", "database")
            .append("likes", 200)
            .append("url", "http://www.tutorialspoint.com/rethinkdb/")
            .append("by", "tutorials point");
            List<Document> list = new ArrayList<Document>();
            list.add(document1);
            list.add(document2);
            collection.insertMany(list);
            // Getting the iterable object
            FindIterable<Document> iterDoc = collection.find();
            int i = 1;
            // Getting the iterator
            Iterator it = iterDoc.iterator();
            while (it.hasNext()) {
                    System.out.println(it.next());
                    i++;
            }
        }
}
```

On compiling, the above program gives you the following result −

```
Connected to the database successfully
Collection sampleCollection selected successfully
Document{{_id=5dce4e9ff68a9c2449e197b2, title=MongoDB, description=database, likes=100, url=http://www.t
Document{{_id=5dce4e9ff68a9c2449e197b3, title=RethinkDB, description=database, likes=200, url=http://www
```

Update Document

To update a document from the collection, **updateOne()** method of **com.mongodb.client.MongoCollection** class is used.

Following is the program to select the first document −

```
import com.mongodb.client.FindIterable;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import com.mongodb.client.model.Filters;
import com.mongodb.client.model.Updates;
import java.util.Iterator;
import org.bson.Document;
import com.mongodb.MongoClient;
import com.mongodb.MongoCredential;
public class UpdatingDocuments {

   public static void main( String args[] ) {

      // Creating a Mongo client
      MongoClient mongo = new MongoClient( "localhost" , 27017 );

      // Creating Credentials
      MongoCredential credential;
      credential = MongoCredential.createCredential("sampleUser", "myDb
         "password".toCharArray());
      System.out.println("Connected to the database successfully");

      // Accessing the database
      MongoDatabase database = mongo.getDatabase("myDb");
      // Retrieving a collection
      MongoCollection<Document> collection = database.getCollection("sa
      System.out.println("Collection myCollection selected successfully
      collection.updateOne(Filters.eq("title", 1), Updates.set("likes",
      System.out.println("Document update successfully...");

      // Retrieving the documents after updation
      // Getting the iterable object
      FindIterable<Document> iterDoc = collection.find();
      int i = 1;
      // Getting the iterator
      Iterator it = iterDoc.iterator();
      while (it.hasNext()) {
         System.out.println(it.next());
         i++;
      }
   }
}
```

Delete a Document

To delete a document from the collection, you need to use the **deleteOne()** method of the **com.mongodb.client.MongoCollection** class.

Following is the program to delete a document −

```
import com.mongodb.client.FindIterable;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import com.mongodb.client.model.Filters;
import java.util.Iterator;
import org.bson.Document;
import com.mongodb.MongoClient;
import com.mongodb.MongoCredential;
public class DeletingDocuments {

   public static void main( String args[] ) {

      // Creating a Mongo client
      MongoClient mongo = new MongoClient( "localhost" , 27017 );

      // Creating Credentials
      MongoCredential credential;
      credential = MongoCredential.createCredential("sampleUser", "myDb",
         "password".toCharArray());
      System.out.println("Connected to the database successfully");

      // Accessing the database
      MongoDatabase database = mongo.getDatabase("myDb");
      // Retrieving a collection
      MongoCollection<Document> collection = database.getCollection("sampleCollection");
      System.out.println("Collection sampleCollection selected successfully");
      // Deleting the documents
      collection.deleteOne(Filters.eq("title", "MongoDB"));
      System.out.println("Document deleted successfully...");
```

```
      // Retrieving the documents after updation
      // Getting the iterable object
      FindIterable<Document> iterDoc = collection.find();
      int i = 1;
      // Getting the iterator
      Iterator it = iterDoc.iterator();
      while (it.hasNext()) {
         System.out.println(it.next());
         i++;
      }
   }
}
```

On compiling, the above program gives you the following result −

```
Connected to the database successfully
Collection sampleCollection selected successfully
Document deleted successfully...
Document{{_id=5dce4e9ff68a9c2449e197b3, title=RethinkDB, description=database, likes=200, url=http://www
```

Dropping a Collection

To drop a collection from a database, you need to use the **drop()** method of the **com.mongodb.client.MongoCollection** class.

Following is the program to delete a collection −

```
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import org.bson.Document;
import com.mongodb.MongoClient;
import com.mongodb.MongoCredential;
public class DropingCollection {

   public static void main( String args[] ) {
      // Creating a Mongo client
      MongoClient mongo = new MongoClient( "localhost" , 27017 );
      // Creating Credentials
      MongoCredential credential;
      credential = MongoCredential.createCredential("sampleUser", "myDb",
         "password".toCharArray());
      System.out.println("Connected to the database successfully");

      // Accessing the database
      MongoDatabase database = mongo.getDatabase("myDb");

      // Creating a collection
      System.out.println("Collections created successfully");
      // Retrieving a collection
      MongoCollection<Document> collection = database.getCollection("sampleCollection");
      // Dropping a Collection
      collection.drop();
      System.out.println("Collection dropped successfully");
   }
}
```

Listing All the Collections

To list all the collections in a database, you need to use the **listCollectionNames()** method of the **com.mongodb.client.MongoDatabase** class.

Following is the program to list all the collections of a database −

```
import com.mongodb.client.MongoDatabase;
import com.mongodb.MongoClient;
import com.mongodb.MongoCredential;
public class ListOfCollection {

   public static void main( String args[] ) {

      // Creating a Mongo client
      MongoClient mongo = new MongoClient( "localhost" , 27017 );
      // Creating Credentials
      MongoCredential credential;
      credential = MongoCredential.createCredential("sampleUser", "myDb",
         "password".toCharArray());
      System.out.println("Connected to the database successfully");

      // Accessing the database
      MongoDatabase database = mongo.getDatabase("myDb");
      System.out.println("Collection created successfully");
      for (String name : database.listCollectionNames()) {
         System.out.println(name);
      }
   }
}
```

MONGODB PHP

To use MongoDB with PHP, you need to use MongoDB PHP driver. Download the driver from the url Download PHP Driver. Make sure to download the latest release of it. Now unzip the archive and put php_mongo.dll in your PHP extension directory ("ext" by default) and add the following line to your php.ini file −

```
extension = php_mongo.dll
```

Make a Connection and Select a Database

To make a connection, you need to specify the database name, if the database doesn't exist then MongoDB creates it automatically.

Following is the code snippet to connect to the database −

```
<?php
   // connect to mongodb
   $m = new MongoClient();

   echo "Connection to database successfully";
   // select a database
   $db = $m->mydb;

   echo "Database mydb selected";
?>
```

When the program is executed, it will produce the following result −

```
Connection to database successfully
Database mydb selected
```

Create a Collection

Following is the code snippet to create a collection −

```
<?php
   // connect to mongodb
   $m = new MongoClient();
   echo "Connection to database successfully";

   // select a database
   $db = $m->mydb;
   echo "Database mydb selected";
   $collection = $db->createCollection("mycol");
   echo "Collection created succsessfully";
?>
```

When the program is executed, it will produce the following result −

```
Connection to database successfully
Database mydb selected
Collection created succsessfully
```

Insert a Document

To insert a document into MongoDB, **insert()** method is used.

Following is the code snippet to insert a document −

```
<?php
   // connect to mongodb
   $m = new MongoClient();
   echo "Connection to database successfully";

   // select a database
   $db = $m->mydb;
   echo "Database mydb selected";
   $collection = $db->mycol;
   echo "Collection selected succsessfully";

   $document = array(
      "title" => "MongoDB",
      "description" => "database",
      "likes" => 100,
      "url" => "http://www.tutorialspoint.com/mongodb/",
      "by" => "tutorials point"
   );

   $collection->insert($document);
   echo "Document inserted successfully";
?>
```

When the program is executed, it will produce the following result −

```
Connection to database successfully
Database mydb selected
Collection selected succsessfully
Document inserted successfully
```

Find All Documents

To select all documents from the collection, find() method is used.

Following is the code snippet to select all documents −

```
<?php
   // connect to mongodb
   $m = new MongoClient();
   echo "Connection to database successfully";

   // select a database
   $db = $m->mydb;
   echo "Database mydb selected";
   $collection = $db->mycol;
   echo "Collection selected succsessfully";
   $cursor = $collection->find();
   // iterate cursor to display title of documents

   foreach ($cursor as $document) {
      echo $document["title"] . "\n";
   }
?>
```

When the program is executed, it will produce the following result −

```
Connection to database successfully
Database mydb selected
Collection selected succsessfully {
   "title": "MongoDB"
}
```

Update a Document

To update a document, you need to use the update() method.

In the following example, we will update the title of inserted document to **MongoDB Tutorial**. Following is the code snippet to update a document −

```
<?php
   // connect to mongodb
   $m = new MongoClient();
   echo "Connection to database successfully";

   // select a database
   $db = $m->mydb;
   echo "Database mydb selected";
   $collection = $db->mycol;
   echo "Collection selected succsessfully";
   // now update the document
   $collection->update(array("title"=>"MongoDB"),
      array('$set'=>array("title"=>"MongoDB Tutorial")));
   echo "Document updated successfully";

   // now display the updated document
   $cursor = $collection->find();

   // iterate cursor to display title of documents
   echo "Updated document";

   foreach ($cursor as $document) {
      echo $document["title"] . "\n";
   }
?>
```

Delete a Document

To delete a document, you need to use remove() method.

In the following example, we will remove the documents that has the title **MongoDB Tutorial**. Following is the code snippet to delete a document −

```
<?php
   // connect to mongodb
   $m = new MongoClient();
   echo "Connection to database successfully";

   // select a database
   $db = $m->mydb;
   echo "Database mydb selected";
   $collection = $db->mycol;
   echo "Collection selected succsessfully";

   // now remove the document
   $collection->remove(array("title"=>"MongoDB Tutorial"),false);
   echo "Documents deleted successfully";

   // now display the available documents
   $cursor = $collection->find();

   // iterate cursor to display title of documents
   echo "Updated document";

   foreach ($cursor as $document) {
      echo $document["title"] . "\n";
   }
?>
```

MONGODB AGGREGATION

The **aggregate** function accepts an array of data transformations which are applied to the data in the order they're defined. This makes aggregation a lot like other data flow pipelines: the transformations that are defined first will be executed first and the result will be used by the next transformation in the sequence.

MongoDB Aggregation Pipeline

Firstly, MongoDB aggregation is a pipeline that processes the data on each pipeline stage. Each stage returns the output, which turns into the input for the next pipeline in the stage.

Here, Data is passed in each pipeline which filters, group and sort the data and returns the result.

```
pipeline = [
  { $match : { … },
  { $group : { … },
  { $sort : { … },
  ...
]
db.collectionName.aggregate(pipeline, options)
```

MATCH OPERATOR EXAMPLE

```
db.test
  .aggregate([
    {
      $match: {
        state: "MA",
      },
    },
  ])
  .pretty()
```

```
> db.test.aggregate([
...     {
...       $match: {
...         state: 'MA'
...       }
...     }
...   ]).pretty();
{
        "_id" : "01001",
        "city" : "AGAWAM",
        "loc" : [
                -72.622739
                42.070206
        ],
        "pop" : 15338,
        "state" : "MA"
}
{
        "_id" : "01012",
        "city" : "CHESTERF
        "loc" : [
                -72.833309
                42.38167
        ],
        "pop" : 177,
        "state" : "MA"
}
{
        "_id" : "01013",
        "city" : "CHICOPEE
        "loc" : [
                -72.607962
                42.162046
        ],
        "pop" : 23396,
        "state" : "MA"
```

GROUP OPERATOR EXAMPLE

```
db.test.aggregate([
  {
    $match: {
      state: "MA",
    },
  },
  {
    $group: {
      _id: "$city",
    },
  },
])
```

```
> db.test.aggregate([
...     {
...       $match: {
...         state: 'MA'
...       }
...     },
...     {
...       $group: {
...         _id: '$city'
...       }
...     }
...   ])
{ "_id" : "ROCHESTER" }
{ "_id" : "RAYNHAM" }
{ "_id" : "PLAINVILLE" }
{ "_id" : "NORTH ATTLEBORO" }
{ "_id" : "NORTH DARTMOUTH" }
{ "_id" : "ACUSHNET" }
{ "_id" : "MATTAPOISETT" }
{ "_id" : "TRURO" }
{ "_id" : "FALL RIVER" }
{ "_id" : "DIGHTON" }
{ "_id" : "EAST FREETOWN" }
{ "_id" : "ATTLEBORO" }
{ "_id" : "CUTTYHUNK" }
{ "_id" : "BASS RIVER" }
{ "_id" : "SEEKONK" }
{ "_id" : "ASSONET" }
{ "_id" : "PADANARAM VILLAG" }
{ "_id" : "PROVINCETOWN" }
{ "_id" : "ORLEANS" }
{ "_id" : "OSTERVILLE" }
Type "it" for more
>
```

Here when we started grouping we are only getting the field that we grouped, but we need to get every other field. So for that we need to provide accumulator expression to retrieve the result of the grouping pipeline.

Accumulator expressions:

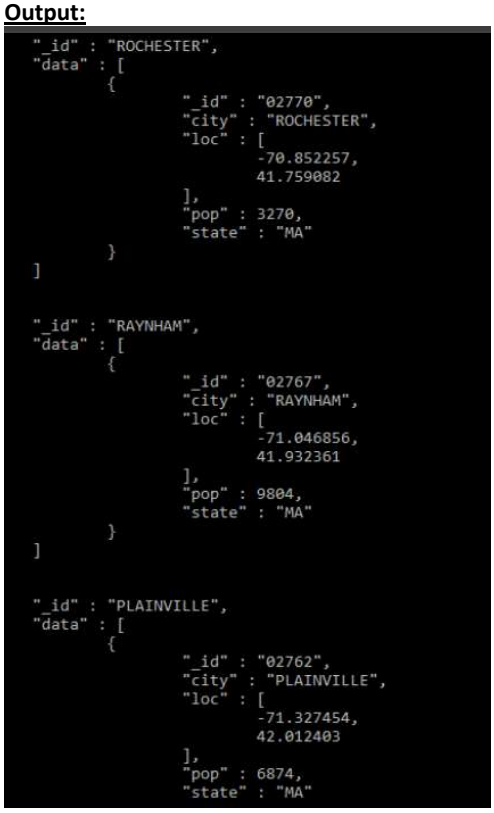
\$first - this expression will return the first document of grouping result
\$push - it will push all the documents into an array based on grouping result
\$max - returns the highest value from the grouping documents.
\$sum - returns the sum(numerial value) of the grouping documents.

$push Example:

Below on the following code you can see we grouped the data based on the id and adding a new column called data which contains the array of roots. (Root mean all the columns of that table)

```
db.test
  .aggregate([
    {
      $match: {
        state: "MA",
      },
    },
    {
      $group: {
        _id: "$city",
        data: {
          $push: "$$ROOT",
        },
      },
    },
  ])
  .pretty()
```

Output:



```
"_id" : "ROCHESTER",
"data" : [
        {
                "_id" : "02770",
                "city" : "ROCHESTER",
                "loc" : [
                        -70.852257,
                        41.759082
                ],
                "pop" : 3270,
                "state" : "MA"
        }
]


"_id" : "RAYNHAM",
"data" : [
        {
                "_id" : "02767",
                "city" : "RAYNHAM",
                "loc" : [
                        -71.046856,
                        41.932361
                ],
                "pop" : 9804,
                "state" : "MA"
        }
]


"_id" : "PLAINVILLE",
"data" : [
        {
                "_id" : "02762",
                "city" : "PLAINVILLE",
                "loc" : [
                        -71.327454,
                        42.012403
                ],
                "pop" : 6874,
                "state" : "MA"
```

$project operator:

- If we want to exhibit only some specific fields on the BSON then we use project operator

```
db.test
  .aggregate([
    {
      $match: {
        state: "MA",
      },
    },
    {
      $group: {
        _id: "$city",
        data: {
          $push: "$$ROOT",
        },
      },
    },
    {
      $project: {
        _id: 0,
        "data.loc": 1,
      },
    },
  ])
  .pretty()
```

```
 "data" : [ { "loc" : [ -70.852257, 41.759082 ] } ] }
 "data" : [ { "loc" : [ -71.046856, 41.932361 ] } ] }
 "data" : [ { "loc" : [ -71.327454, 42.012403 ] } ] }
 "data" : [ { "loc" : [ -71.329757, 41.977542 ] }, { "loc" : [ -71.310353, 41.970979 ] } ] }
 "data" : [ { "loc" : [ -70.995769, 41.633789 ] } ] }
 "data" : [ { "loc" : [ -70.908652, 41.6997 ] } ] }
 "data" : [ { "loc" : [ -70.816357, 41.661845 ] } ] }
 "data" : [ { "loc" : [ -70.056362, 41.998792 ] } ] }
 "data" : [ { "loc" : [ -71.174822, 41.684975 ] }, { "loc" : [ -71.157424, 41.688305 ] }, { "
12 ] } ] }
 "data" : [ { "loc" : [ -71.142723, 41.812505 ] } ] }
 "data" : [ { "loc" : [ -70.967709, 41.763455 ] } ] }
 "data" : [ { "loc" : [ -71.30092, 41.929599 ] } ] }
 "data" : [ { "loc" : [ -70.87854, 41.443601 ] } ] }
 "data" : [ { "loc" : [ -70.19731, 41.672805 ] } ] }
 "data" : [ { "loc" : [ -71.322406, 41.837835 ] } ] }
 "data" : [ { "loc" : [ -71.060736, 41.797458 ] } ] }
 "data" : [ { "loc" : [ -70.956521, 41.591728 ] } ] }
 "data" : [ { "loc" : [ -70.186504, 42.053364 ] } ] }
 "data" : [ { "loc" : [ -69.982198, 41.779161 ] } ] }
 "data" : [ { "loc" : [ -70.383726, 41.63005 ] } ] }
ype "it" for more
```

Sort operator

```
db.test.aggregate([
  {
    $match: {
      state: "MA",
    },
  },
  {
    $sort: {
      pop: 1,
    },
  },
])
```

- For 1 it interpreted as ascending and -1 it is descending

```
{ "_id" : "02163", "city" : "CAMBRIDGE", "loc" : [ -71.141879, 42.364005 ], "pop" : 0, "state" : "MA" }
{ "_id" : "01338", "city" : "BUCKLAND", "loc" : [ -72.764124, 42.615174 ], "pop" : 16, "state" : "MA" }
{ "_id" : "01350", "city" : "MONROE", "loc" : [ -72.960156, 42.723885 ], "pop" : 97, "state" : "MA" }
{ "_id" : "02713", "city" : "CUTTYHUNK", "loc" : [ -70.87854, 41.443601 ], "pop" : 98, "state" : "MA" }
{ "_id" : "01032", "city" : "GOSHEN", "loc" : [ -72.844092, 42.466234 ], "pop" : 122, "state" : "MA" }
{ "_id" : "01258", "city" : "SOUTH EGREMONT", "loc" : [ -73.456575, 42.101153 ], "pop" : 135, "state" : "MA" }
{ "_id" : "01346", "city" : "HEATH", "loc" : [ -72.839101, 42.685347 ], "pop" : 174, "state" : "MA" }
{ "_id" : "01012", "city" : "CHESTERFIELD", "loc" : [ -72.833309, 42.38167 ], "pop" : 177, "state" : "MA" }
{ "_id" : "02210", "city" : "BOSTON", "loc" : [ -71.046511, 42.348921 ], "pop" : 308, "state" : "MA" }
{ "_id" : "01245", "city" : "WEST OTIS", "loc" : [ -73.213452, 42.187847 ], "pop" : 329, "state" : "MA" }
{ "_id" : "01243", "city" : "MIDDLEFIELD", "loc" : [ -73.006226, 42.34795 ], "pop" : 384, "state" : "MA" }
{ "_id" : "01379", "city" : "WENDELL", "loc" : [ -72.400851, 42.565644 ], "pop" : 393, "state" : "MA" }
{ "_id" : "01355", "city" : "NEW SALEM", "loc" : [ -72.306241, 42.514643 ], "pop" : 456, "state" : "MA" }
{ "_id" : "01745", "city" : "SOUTHBOROUGH", "loc" : [ -71.502256, 42.293221 ], "pop" : 506, "state" : "MA" }
{ "_id" : "01222", "city" : "ASHLEY FALLS", "loc" : [ -73.320195, 42.059552 ], "pop" : 561, "state" : "MA" }
{ "_id" : "01070", "city" : "PLAINFIELD", "loc" : [ -72.918289, 42.514393 ], "pop" : 571, "state" : "MA" }
{ "_id" : "01071", "city" : "RUSSELL", "loc" : [ -72.840343, 42.147063 ], "pop" : 608, "state" : "MA" }
{ "_id" : "01259", "city" : "SOUTHFIELD", "loc" : [ -73.260933, 42.078014 ], "pop" : 622, "state" : "MA" }
{ "_id" : "01367", "city" : "ROWE", "loc" : [ -72.925776, 42.695289 ], "pop" : 630, "state" : "MA" }
{ "_id" : "01256", "city" : "SAVOY", "loc" : [ -73.023281, 42.576964 ], "pop" : 632, "state" : "MA" }
Type "it" for more
```


\$limit operator

Limit operator limits the number of documents retrieved from the database.

```
db.test.aggregate([
  {
    $match: {
      state: "MA",
    },
  },
  {
    $sort: {
      pop: 1,
    },
  },
  {
    $limit: 5,
  },
])
```

So, the above query will return only five documents from the database.


```
{ "_id" : "02163", "city" : "CAMBRIDGE", "loc" : [ -71.141879, 42.364005 ], "pop" : 0, "state" : "MA" }
{ "_id" : "01338", "city" : "BUCKLAND", "loc" : [ -72.764124, 42.615174 ], "pop" : 16, "state" : "MA" }
{ "_id" : "01350", "city" : "MONROE", "loc" : [ -72.960156, 42.723885 ], "pop" : 97, "state" : "MA" }
{ "_id" : "02713", "city" : "CUTTYHUNK", "loc" : [ -70.87854, 41.443601 ], "pop" : 98, "state" : "MA" }
{ "_id" : "01032", "city" : "GOSHEN", "loc" : [ -72.844092, 42.466234 ], "pop" : 122, "state" : "MA" }
>
```


\$addFields operator

Sometimes you need to create a custom field based on the aggregated data. you can achieve this using **\$addField** operator.

```
db.test.aggregate([
  {
    $match: {
      state: "MA",
    },
  },
  {
    $addFields: {
      stateAlias: "MAS",
    },
  },
])
```

```
{ "_id" : "01001", "city" : "AGAWAM", "loc" : [ -72.622739, 42.070206 ], "pop" : 15338, "state" : "MA", "stateAlias" : "MAS" }
{ "_id" : "01012", "city" : "CHESTERFIELD", "loc" : [ -72.833309, 42.38167 ], "pop" : 177, "state" : "MA", "stateAlias" : "MAS" }
{ "_id" : "01013", "city" : "CHICOPEE", "loc" : [ -72.607962, 42.162046 ], "pop" : 23396, "state" : "MA", "stateAlias" : "MAS" }
{ "_id" : "01020", "city" : "CHICOPEE", "loc" : [ -72.576142, 42.176443 ], "pop" : 31495, "state" : "MA", "stateAlias" : "MAS" }
{ "_id" : "01022", "city" : "WESTOVER AFB", "loc" : [ -72.558657, 42.196672 ], "pop" : 1764, "state" : "MA", "stateAlias" : "MAS" }
{ "_id" : "01026", "city" : "CUMMINGTON", "loc" : [ -72.905767, 42.435296 ], "pop" : 1484, "state" : "MA", "stateAlias" : "MAS" }
{ "_id" : "01028", "city" : "EAST LONGMEADOW", "loc" : [ -72.505565, 42.067203 ], "pop" : 13367, "state" : "MA", "stateAlias" : "MAS"
}
{ "_id" : "01030", "city" : "FEEDING HILLS", "loc" : [ -72.675077, 42.07182 ], "pop" : 11985, "state" : "MA", "stateAlias" : "MAS" }
{ "_id" : "01027", "city" : "MOUNT TOM", "loc" : [ -72.679921, 42.264319 ], "pop" : 16864, "state" : "MA", "stateAlias" : "MAS" }
{ "_id" : "01032", "city" : "GOSHEN", "loc" : [ -72.844092, 42.466234 ], "pop" : 122, "state" : "MA", "stateAlias" : "MAS" }
{ "_id" : "01031", "city" : "GILBERTVILLE", "loc" : [ -72.198585, 42.332194 ], "pop" : 2385, "state" : "MA", "stateAlias" : "MAS" }
{ "_id" : "01033", "city" : "GRANBY", "loc" : [ -72.520001, 42.255704 ], "pop" : 5526, "state" : "MA", "stateAlias" : "MAS" }
{ "_id" : "01034", "city" : "TOLLAND", "loc" : [ -72.908793, 42.070234 ], "pop" : 1652, "state" : "MA", "stateAlias" : "MAS" }
{ "_id" : "01035", "city" : "HADLEY", "loc" : [ -72.571499, 42.36062 ], "pop" : 4231, "state" : "MA", "stateAlias" : "MAS" }
{ "_id" : "01038", "city" : "HATFIELD", "loc" : [ -72.616735, 42.38439 ], "pop" : 3184, "state" : "MA", "stateAlias" : "MAS" }
{ "_id" : "01036", "city" : "HAMPDEN", "loc" : [ -72.431823, 42.064756 ], "pop" : 4709, "state" : "MA", "stateAlias" : "MAS" }
{ "_id" : "01039", "city" : "HAYDENVILLE", "loc" : [ -72.703178, 42.381799 ], "pop" : 1387, "state" : "MA", "stateAlias" : "MAS" }
{ "_id" : "01050", "city" : "HUNTINGTON", "loc" : [ -72.873341, 42.265301 ], "pop" : 2084, "state" : "MA", "stateAlias" : "MAS" }
{ "_id" : "01040", "city" : "HOLYOKE", "loc" : [ -72.626193, 42.202007 ], "pop" : 43704, "state" : "MA", "stateAlias" : "MAS" }
{ "_id" : "01008", "city" : "BLANDFORD", "loc" : [ -72.936114, 42.182949 ], "pop" : 1240, "state" : "MA", "stateAlias" : "MAS" }
Type "it" for more
```


\$lookup operator

After that, lookup is one of the popular aggregation operators in mongodb. if you are from SQL background. you can relate this with JOIN Query in RDBMS.

```
db.universities
  .aggregate([
    { $match: { name: "USAL" } },
    { $project: { _id: 0, name: 1 } },
    {
      $lookup: {
        from: "courses",
        localField: "name",
        foreignField: "university",
        as: "courses",
      },
    },
  ])
  .pretty()
```

- **from** - it takes the collection that it wants to perform the join.
- **localField** - it specifies the field from the input document. Here, it takes the field **name** from the **universities** collection.
- **foreignField** - it specifies the field from the collection that it performs the join. Here, it is a **university** field from **courses** collection.
- **as** - it specifies the alias for the field name.

Example:

```
> db.customers.aggregate([
  { $match: { "zip": 90210 }}
]);
```

This will return the array of customers that live in the 90210 zip code. Using the **match** stage in this way is no different from using the **find** method on a collection. Let's see what kind of insights we can gather by adding some stages to the pipeline.

Example:

```
> db.customers.aggregate([
  { $match: {"zip": "90210"}},
  {
    $group: {
      _id: null,
      count: {
        $sum: 1
      }
    }
  }
]);
```

The `$group` transformation allows us to group documents together and performs transformations or operations across all of those grouped documents. In this case, we're creating a new field in the results called *count* which adds 1 to a running sum for every document. The `_id` field is required for grouping and would normally contain fields from each document that we'd like to preserve (ie: phoneNumber). Since we're just looking for the count of every document, we can make it null here.

```
{ "_id" : null, "count" : 24 }
```

Example:

Let's start by calculating the total amount of sales made for the month of January. We'll start by matching only transactions that occurred between January 1 and January 31.

```
{
   $match: {
    transactionDate: {
        $gte: ISODate("2017-01-01T00:00:00.000Z"),
        $lt: ISODate("2017-02-01T00:00:00.000Z")
    }
  }
}
```

Example:

The next stage of the pipeline is summing the transaction amounts and putting that amount in a new field called `total`:

```
{
  $group: {
    _id: null,
    total: {
      $sum: "$amount"
    }
  }
}
```

Final Query:

```
> db.transactions.aggregate([
  {
    $match: {
      transactionDate: {
        $gte: ISODate("2017-01-01T00:00:00.000Z"),
        $lt: ISODate("2017-01-31T23:59:59.000Z")
      }
    }
  }, {
    $group: {
      _id: null,
      total: {
        $sum: "$amount"
      }
    }
  }
]);
```

```
{ _id: null, total: 20333.00 }
```

- If we write group after match we no need to mention condition to group , it takes match output as the input for the group and if we use any aggregate functions in the group those got printed as the output considering whole dataset.

```
> db.transactions.aggregate([
  {
    $match: {
      transactionDate: {
        $gte: ISODate("2017-01-01T00:00:00.000Z"),
        $lt: ISODate("2017-01-31T23:59:59.000Z")
      }
    }
  }, {
    $group: {
      _id: null,
      total: {
        $sum: "$amount"
      },
      average_transaction_amount: {
        $avg: "$amount"
      },
      min_transaction_amount: {
        $min: "$amount"
      },
      max_transaction_amount: {
        $max: "$amount"
      }
    }
  }
]);
```

Output:

```
{
  _id: null,
  total: 20333.00,
  average_transaction_amount: 8.50,
  min_transaction_amount: 2.99,
  max_transaction_amount: 347.22
}
```

Complete Mongo Advance

Thursday, March 24, 2022 3:32 PM

MONGODB RELATIONSHIPS

- Relationships in MongoDB represent how various documents are logically related to each other. Relationships can be modeled via Embedded and Referenced approaches. Such relationships can be either 1:1, 1:N, N:1 or N:N.

Modeling Embedded Relationships

In the embedded approach, we will embed the address document inside the user document.

```
> db.users.insert({
        {
                "_id":ObjectId("52ffc33cd85242f436000001"),
                "contact": "987654321",
                "dob": "01-01-1991",
                "name": "Tom Benzamin",
                "address": [
                        {
                                "building": "22 A, Indiana Apt",
                                "pincode": 123456,
                                "city": "Los Angeles",
                                "state": "California"
                        },
                        {
                                "building": "170 A, Acropolis Apt",
                                "pincode": 456789,
                                "city": "Chicago",
                                "state": "Illinois"
                        }
                ]
        }
})
```

This approach maintains all the related data in a single document, which makes it easy to retrieve and maintain. The whole document can be retrieved in a single query such as −

```
>db.users.findOne({"name":"Tom Benzamin"},{"address":1})
```

Note that in the above query, **db** and **users** are the database and collection respectively.

The drawback is that if the embedded document keeps on growing too much in size, it can impact the read/write performance.

Modeling Referenced Relationships

This is the approach of designing normalized relationship. In this approach, both the user and address documents will be maintained separately but the user document will contain a field that will reference the address document's **id** field.

```
{
   "_id":ObjectId("52ffc33cd85242f436000001"),
   "contact": "987654321",
   "dob": "01-01-1991",
   "name": "Tom Benzamin",
   "address_ids": [
      ObjectId("52ffc4a5d85242602e000000"),
      ObjectId("52ffc4a5d85242602e000001")
   ]
}
```

As shown above, the user document contains the array field **address_ids** which contains ObjectIds of corresponding addresses. Using these ObjectIds, we can query the address documents and get address details from there. With this approach, we will need two queries: first to fetch the **address_ids** fields from **user** document and second to fetch these addresses from **address** collection.

```
>var result = db.users.findOne({"name":"Tom Benzamin"},{"address_ids":1})
>var addresses = db.address.find({"_id":{"$in":result["address_ids"]}})
```

MONGODB DATABASE REFERENCES

As seen in the last chapter of MongoDB relationships, to implement a normalized database structure in MongoDB, we use the concept of **Referenced Relationships** also referred to as **Manual References** in which we manually store the referenced document's id inside other document. However, in cases where a document contains references from different collections, we can use **MongoDB DBRefs**.

DBRefs vs Manual References

As an example scenario, where we would use DBRefs instead of manual references, consider a database where we are storing different types of addresses (home, office, mailing, etc.) in different collections (address_home, address_office, address_mailing, etc). Now, when a **user** collection's document references an address, it also needs to specify which collection to look into based on the address type. In such scenarios where a document references documents from many collections, we should use DBRefs.

Using DBRefs

There are three fields in DBRefs −

- **$ref** − This field specifies the collection of the referenced document
- **$id** − This field specifies the _id field of the referenced document
- **$db** − This is an optional field and contains the name of the database in which the referenced document lies

Consider a sample user document having DBRef field **address** as shown in the code snippet −

```
{
   "_id":ObjectId("53402597d852426020000002"),
   "address": {
   "$ref": "address_home",
   "$id": ObjectId("534009e4d852427820000002"),
   "$db": "tutorialspoint"},
   "contact": "987654321",
   "dob": "01-01-1991",
   "name": "Tom Benzamin"
}
```

The **address** DBRef field here specifies that the referenced address document lies in **address_home** collection under **tutorialspoint** database and has an id of 534009e4d852427820000002.

The following code dynamically looks in the collection specified by **$ref** parameter (**address_home** in our case) for a document with id as specified by **$id** parameter in DBRef.

```
>var user = db.users.findOne({"name":"Tom Benzamin"})
>var dbRef = user.address
>db[dbRef.$ref].findOne({"_id":(dbRef.$id)})
```

The above code returns the following address document present in **address_home** collection −

```
{
   "_id" : ObjectId("534009e4d852427820000002"),
   "building" : "22 A, Indiana Apt",
   "pincode" : 123456,
   "city" : "Los Angeles",
   "state" : "California"
}
```

Covered Query

What is a Covered Query?

As per the official MongoDB documentation, a covered query is a query in which −

- All the fields in the query are part of an index.
- All the fields returned in the query are in the same index.

Since all the fields present in the query are part of an index, MongoDB matches the query conditions and returns the result using the same index without actually looking inside the documents. Since indexes are present in RAM, fetching data from indexes is much faster as compared to fetching data by scanning documents.

Using Covered Queries

To test covered queries, consider the following document in the **users** collection −

```
{
   "_id": ObjectId("53402597d852426020000003"),
   "contact": "987654321",
   "dob": "01-01-1991",
   "gender": "M",
   "name": "Tom Benzamin",
   "user_name": "tombenzamin"
}
```

We will first create a compound index for the **users** collection on the fields **gender** and **user_name** using the following query −

```
>db.users.createIndex({gender:1,user_name:1})
{
        "createdCollectionAutomatically" : false,
        "numIndexesBefore" : 1,
        "numIndexesAfter" : 2,
        "ok" : 1
}
```

Now, this index will cover the following query −

```
>db.users.find({gender:"M"},{user_name:1,_id:0})
{ "user_name" : "tombenzamin" }
```

That is to say that for the above query, MongoDB would not go looking into database documents. Instead it would fetch the required data from indexed data which is very fast.

Since our index does not include **_id** field, we have explicitly excluded it from result set of our query, as MongoDB by default returns _id field in every query. So the following query would not have been covered inside the index created above −

```
>db.users.find({gender:"M"},{user_name:1})
{ "_id" : ObjectId("53402597d852426020000003"), "user_name" : "tombenzamin" }
```

Lastly, remember that an index cannot cover a query if −

- Any of the indexed fields is an array
- Any of the indexed fields is a subdocument

Advanced Indexing

General ways of doing indexing

we have inserted the following document in the collection named users as shown below −

```
db.users.insert(
        {
                "address": {
                        "city": "Los Angeles",
                        "state": "California",
                        "pincode": "123"
                },
                "tags": [
                        "music",
                        "cricket",
                        "blogs"
                ],
                "name": "Tom Benzamin"
        }
)
```

The above document contains an **address sub-document** and a **tags array**.

- Now if we want to access the tags index wise then we can create an index on the field of the document and utilize the sub fields of that using separate indexes.
- Now let's create an index on tags field of the document. So we can access music , cricket and blogs with their respective indexes.

```
>db.users.createIndex({"tags":1})
{
"createdCollectionAutomatically" : false,
"numIndexesBefore" : 2,
"numIndexesAfter" : 3,
"ok" : 1
}
```

- After creating the index if we try to access the document with respective to tag then we get the following output. And if we want to get the proper output then we have to use explain() method

```
> db.users.find({tags:"cricket"}).pretty()
{
        "_id" : ObjectId("5dd7c927f1dd4583e7103fdf"),
        "address" : {
                "city" : "Los Angeles",
                "state" : "California",
                "pincode" : "123"
        },
        "tags" : [
                "music",
                "cricket",
                "blogs"
        ],
        "name" : "Tom Benzamin"
}
>
```

To verify that proper indexing is used, use the following **explain** command −

```
>db.users.find({tags:"cricket"}).explain()
```

Indexing sub document fields

Assume the document is in this manner:

```
> db.users.find({"address.city":"Los Angeles"}).pretty()
{
        "_id" : ObjectId("5dd7c927f1dd4583e7103fdf"),
        "address" : {
                "city" : "Los Angeles",
                "state" : "California",
                "pincode" : "123"
        },
        "tags" : [
                "music",
                "cricket",
                "blogs"
        ],
        "name" : "Tom Benzamin"
}
```

Now we need to create indexes of sub document fields like city , state, pincode , so we can use text search operation on that.

```
>db.users.createIndex({"address.city":1,"address.state":1,"address.pincode":1})
{
        "numIndexesBefore" : 4,
        "numIndexesAfter" : 4,
        "note" : "all indexes already exist",
        "ok" : 1
}
```

The following is the application of text search on the sub document fields.

```
>db.users.find({"address.city":"Los Angeles","address.state":"California"}).pretty()
{
        "_id" : ObjectId("5dd7c927f1dd4583e7103fdf"),
        "address" : {
                "city" : "Los Angeles",
                "state" : "California",
                "pincode" : "123"
        },
        "tags" : [
                "music",
                "cricket",
                "blogs"
        ],
        "name" : "Tom Benzamin"
}
```

Extra Overhead

Every index occupies some space as well as causes an overhead on each insert, update and delete. So if you rarely use your collection for read operations, it makes sense not to use indexes.

RAM Usage

Since indexes are stored in RAM, you should make sure that the total size of the index does not exceed the RAM limit. If the total size increases the RAM size, it will start deleting some indexes, causing performance loss.

- MongoDB will not create an index if the value of existing index field exceeds the index key limit.

Maximum Ranges

- A collection cannot have more than 64 indexes.
- The length of the index name cannot be longer than 125 characters.
- A compound index can have maximum 31 fields indexed.

Object ID

An **ObjectId** is a 12-byte BSON type having the following structure −

- The first 4 bytes representing the seconds since the unix epoch
- The next 3 bytes are the machine identifier

An **ObjectId** is a 12-byte BSON type having the following structure −

- The first 4 bytes representing the seconds since the unix epoch
- The next 3 bytes are the machine identifier
- The next 2 bytes consists of **process id**
- The last 3 bytes are a random counter value

Create a new Object:

To generate a new ObjectId use the following code −

```
>newObjectId = ObjectId()
```

The above statement returned the following uniquely generated id −

```
ObjectId("5349b4ddd2781d08c09890f3")
```

Instead of MongoDB generating the ObjectId, you can also provide a 12-byte id −

```
>myObjectId = ObjectId("5349b4ddd2781d08c09890f4")
```

- If we want to check when the object is actually generated we can check using the following. Due to this we no need to store the creation time of the document.

```
>ObjectId("5349b4ddd2781d08c09890f4").getTimestamp()
```

This will return the creation time of this document in ISO date format −

```
ISODate("2014-04-12T21:49:17Z")
```

- If we want to get string from the object id then do the following

```
>newObjectId.str
```

The above code will return the string format of the Guid −

```
5349b4ddd2781d08c09890f3
```

- Here newObjectId is the instance created by ObjectID()

MongoDB MapReduce

- Map-reduce is a data processing paradigm for condensing large volumes of data into useful aggregated results.
- MapReduce is generally used for processing large data sets.
- Suppose we are having student data collection with a document like mentioned below and we need to find calculate the sum of marks of all students. We want to perform the operation using map reduce function.

```
> //MapReduce in MongoDB
> //Map
> //Reduce
> db.studentdata.find()
{ "_id" : ObjectId("5f94ae8e78b6fd018d4b9448"), "name" : "Ram", "marks" : 40, "age" : 20 }
{ "_id" : ObjectId("5f94aeb778b6fd018d4b9449"), "name" : "Siya", "marks" : 50, "age" : 20 }
{ "_id" : ObjectId("5f94aeca78b6fd018d4b944a"), "name" : "Swati", "marks" : 30, "age" : 18 }
{ "_id" : ObjectId("5f94aee178b6fd018d4b944b"), "name" : "Rahul", "marks" : 50, "age" : 21 }
{ "_id" : ObjectId("5f94aef678b6fd018d4b944c"), "name" : "Riya", "marks" : 60, "age" : 21 }
> //calculate the sum of marks of all students
> var mapfunction=function(){emit(this.age,this.marks)}
> var reducefunctin=function(key,values){return Array.sum(values)}
> db.studentdata.mapReduce(mapfunction,reducefunction,{'out':'Result1_mapreduce'})
{ "result" : "Result1_mapreduce", "ok" : 1 }
> db.Result1_mapreduce.find()
{ "_id" : 18, "value" : 30 }
{ "_id" : 21, "value" : 110 }
{ "_id" : 20, "value" : 90 }
```

- Here while declaring the map function we are emitting two parameters , one is age and other is marks. The first parameter (age) signifies the on what we are using the group function and the second parameter is the one we are sending for aggregation.
- While coming to reduce function the key and value parameter are the ones that is emitted from previous function.
- In this reduce function we are performing the aggregate task and returning the result to the user defined output parameter Result1_mapreduce. It will automatically become a collection and the results will be tables in form of fields.

Find average of marks of students whose age is greater than 18

```
{ "_id" : 18, "value" : 30 }
{ "_id" : 21, "value" : 110 }
{ "_id" : 20, "value" : 90 }
> db.studentdata.mapReduce(function(){emit(this.age,this.marks);},function(key,values){
... return Array.avg(values)},{query:{age:{$gt:18}},out:'Result2_mapreduce'})
{ "result" : "Result2_mapreduce", "ok" : 1 }
```

- Here we can see the contents of the document
- Vivid to the previous , we are creating a single function where we will first write the map function and then reduce function and then query function which is similar to filter of the database and output that to the external variable.
- Query parameter acts as an filter.

```
{ "_id" : 18, "value" : 30 }
{ "_id" : 21, "value" : 110 }
{ "_id" : 20, "value" : 90 }
> db.studentdata.mapReduce(function(){emit(this.age,this.marks);},function(key,values){
... return Array.avg(values)},{query:{age:{$gt:18}},out:'Result2_mapreduce'})
{ "result" : "Result2_mapreduce", "ok" : 1 }
> db.Result2_mapreduce.find()
{ "_id" : 21, "value" : 55 }
{ "_id" : 20, "value" : 45 }
```