

Mongo DB

MongoDB

- MongoDB is a **open source, cross-platform, document oriented database** that provides, high performance, high availability, and easy scalability.
- MongoDB works on **concept of collection and document**.
- Database is a physical container for collections, Collection is a group of MongoDB documents and a document is a set of key-value pairs.
- MongoDB uses **dynamic schemas**, meaning that you can create records **without first defining the structure**, such as the fields or the types of their values.
- You can **change the structure** of records (which we call documents) simply by adding new fields or deleting existing ones.
- This data model give you the ability to represent hierarchical relationships, to store arrays, and other more complex structures **easily**.

Node.js MongoDB

- Node.js can be used in database applications. Node js can work with both relational (such as Oracle and MS SQL Server, MySQL, IBM DB2) and non-relational databases (such as MongoDB, DocumentDB, Hbase, Neo4j).
- Over the years, NoSQL database as **MongoDB** becomes quite popular as databases for storing data.
- In relational database you need to create the table, define schema, set the data types of fields etc before you can actually insert the data. In NoSQL you don't have to worry about that, you can insert, update data on the fly.
- MongoDB is an **open-source, document database** designed with both scalability and developer agility in mind.

- Instead of storing data in rows and columns as one would with a relational database, MongoDB **stores JSON documents** in collections with dynamic schemas.
- MongoDB is that it supports dynamic schema which means one document of a collection can have 4 fields while the other document has only 3 fields. This is not possible in relational database.
- MongoDB's document data model makes it easy for you to **store and combine data of any structure**, without giving up sophisticated validation rules, flexible data access, and rich indexing functionality.
- **MongoDB Atlas** is a database as a service for MongoDB, letting you focus on apps instead of ops. With MongoDB Atlas, you only pay for what you use with a convenient hourly billing model. With the click of a button, you can scale up and down when you need to, with no downtime, full security, and high performance.

Mapping relational database to MongoDB

RDBMS	MongoDB
Database	Database
Table	Collection
Tuple/Row	Document
column	Field
Table Join	Embedded Documents
Primary Key	Primary Key (Default key _id provided by mongodb itself)
Database Server and Client	
Mysqld/Oracle	mongod
mysql/sqlplus	mongo

Relational databases usually work with structured data, while non-relational databases usually work with semi-structured data (i.e. XML, JSON)

Relational Database

Student_Id	Student_Name	Age	College
1001	Chaitanya	30	Beginnersbook
1002	Steve	29	Beginnersbook
1003	Negan	28	Beginnersbook



MongoDB

```
{
  "_id": ObjectId("....."),
  "Student_Id": 1001,
  "Student_Name": "Chaitanya",
  "Age": 30,
  "College": "Beginnersbook"
}
{
  "_id": ObjectId("....."),
  "Student_Id": 1002,
  "Student_Name": "Steve",
  "Age": 29,
  "College": "Beginnersbook"
}
{
  "_id": ObjectId("....."),
  "Student_Id": 1003,
  "Student_Name": "Negan",
  "Age": 28,
  "College": "Beginnersbook"
}
```

```
{
  name: "Chaitanya", ← Field: Value
  age: 30, ← Field: Value
  website: "beginnersbook.com", ← Field: Value
  hobbies: ["teaching", "watching tv"] ← Field: Value
}
```

} Document

Sample Document

```
{  _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,
  comments: [
    {
      user: 'user1',
      message: 'My first comment',
      dateCreated: new Date(2011,1,20,2,15),
      like: 0
    },
    {
      user: 'user2',
      message: 'My second comments',
      dateCreated: new Date(2011,1,25,7,45),
      like: 5
    }
  ]
}
```

Why use MongoDB instead of MySQL?

- Organizations of **all sizes** are adopting MongoDB because it enables them **to build applications faster**.
- Development is simplified as MongoDB documents map naturally to modern, object-oriented programming languages.
- For example, schema changes that took days or weeks in The Weather Channel's MySQL databases could be made in just hours with MongoDB.
- MongoDB is a general purpose database that is used for a variety of use cases.
- The most common use cases for MongoDB include Internet of Things, Mobile, Real-Time Analytics, Personalization, Catalog, and Content Management.

Differences in Query Languages

- Both MySQL and MongoDB have a rich query language

MySQL

```
INSERT INTO users (user_id, age, status)
VALUES ('bcd001', 45, 'A')
```

```
SELECT * FROM users
```

```
UPDATE users SET status = 'C'
WHERE age > 25
```

MongoDB

```
db.users.insert({
  user_id: 'bcd001',
  age: 45,
  status: 'A'
})
```

```
db.users.find()
```

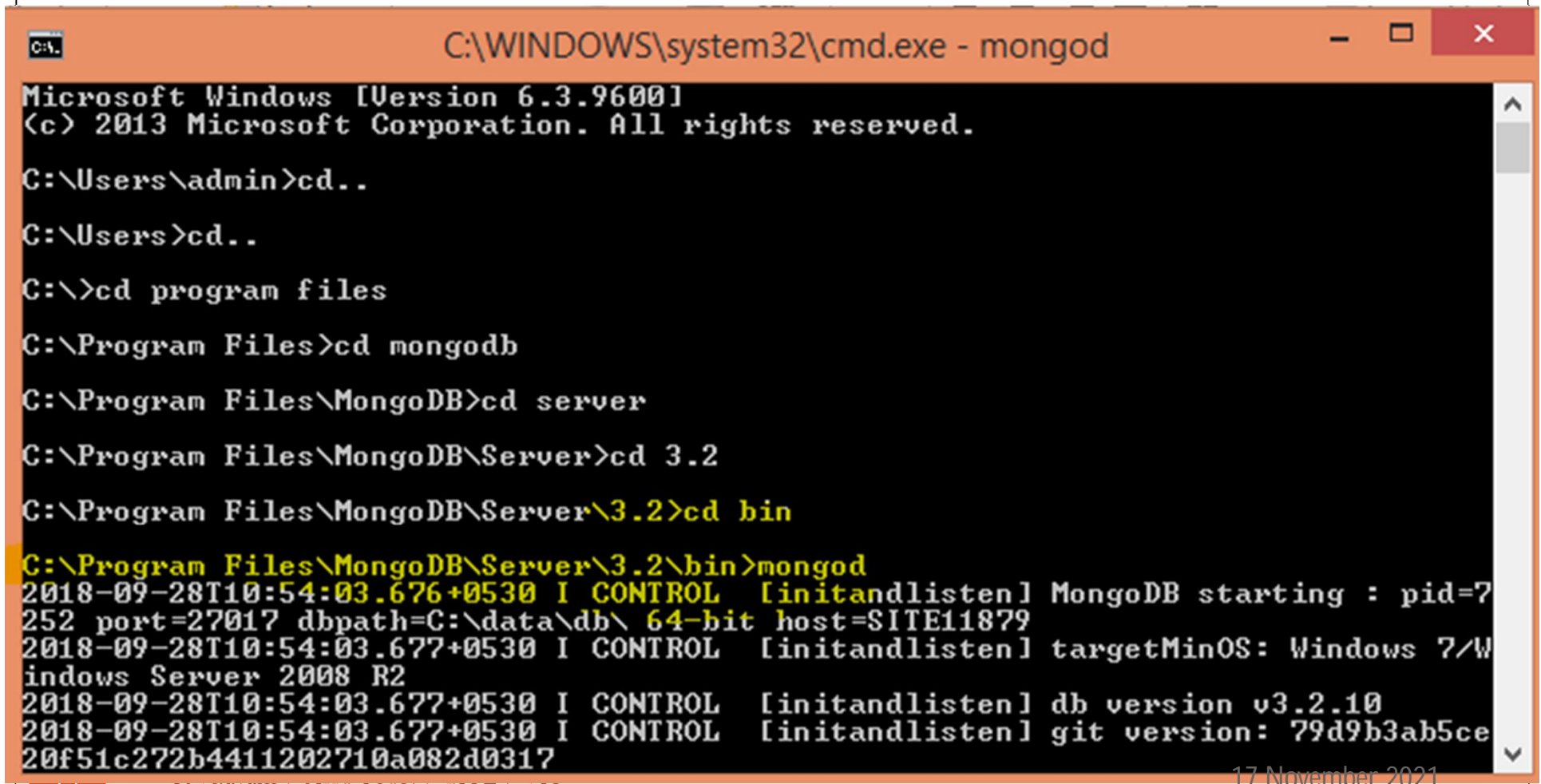
```
db.users.update(
  { age: { $gt: 25 } },
  { $set: { status: 'C' } },
  { multi: true }
)
```

Installation Procedure in Windows

1. Download the latest release of MongoDB from <http://www.mongodb.org/downloads>
2. In this, select community server, then windows 64bit software msi format. (depends on your system)
3. It will be installed automatically in C:\Program Files\MongoDB
4. You should manually create a folder like C:\data\db to store MongoDB files.
5. You should open two command prompt for mongo server and mongo client.
6. C:\Program Files\MongoDB\Server\4.2\bin>mongod.exe --dbpath "C:\data"

Run the command to act as a **mongo server**

C:\Program Files\MongoDB\Server\3.2\bin>mongod



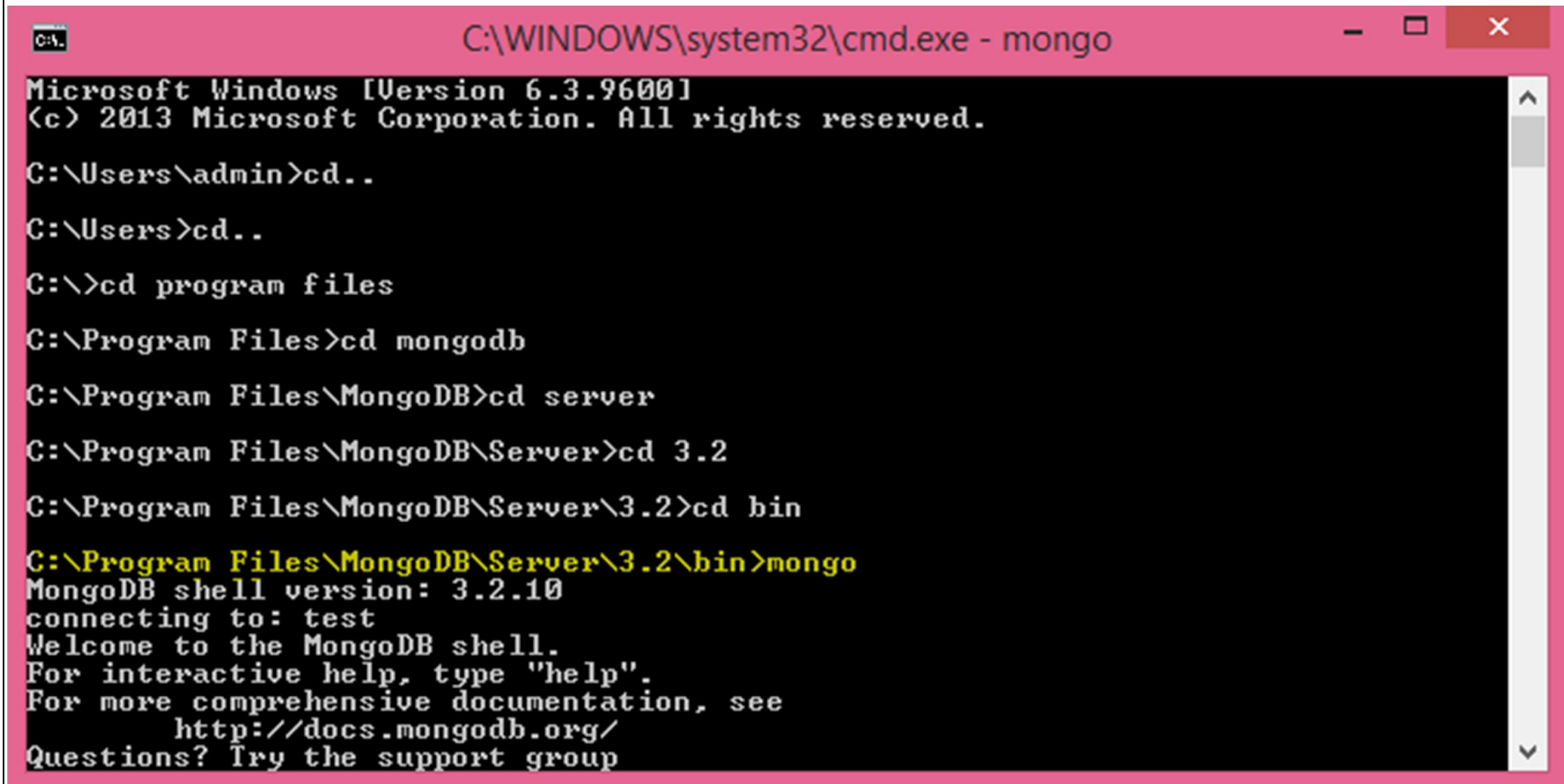
```
C:\WINDOWS\system32\cmd.exe - mongod

Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.

C:\Users\admin>cd..
C:\Users>cd..
C:\>cd program files
C:\Program Files>cd mongodb
C:\Program Files\MongoDB>cd server
C:\Program Files\MongoDB\Server>cd 3.2
C:\Program Files\MongoDB\Server\3.2>cd bin
C:\Program Files\MongoDB\Server\3.2\bin>mongod
2018-09-28T10:54:03.676+0530 I CONTROL [initandlisten] MongoDB starting : pid=7
252 port=27017 dbpath=C:\data\db\ 64-bit host=SITE11879
2018-09-28T10:54:03.677+0530 I CONTROL [initandlisten] targetMinOS: Windows 7/W
indows Server 2008 R2
2018-09-28T10:54:03.677+0530 I CONTROL [initandlisten] db version v3.2.10
2018-09-28T10:54:03.677+0530 I CONTROL [initandlisten] git version: 79d9b3ab5ce
20f51c272b4411202710a082d0317
```

Run the command to act as a **mongo client**

C:\Program Files\MongoDB\Server\3.2\bin>mongo



```
C:\WINDOWS\system32\cmd.exe - mongo
Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.

C:\Users\admin>cd..
C:\Users>cd..
C:\>cd program files
C:\Program Files>cd mongodb
C:\Program Files\MongoDB>cd server
C:\Program Files\MongoDB\Server>cd 3.2
C:\Program Files\MongoDB\Server\3.2>cd bin
C:\Program Files\MongoDB\Server\3.2\bin>mongo
MongoDB shell version: 3.2.10
connecting to: test
Welcome to the MongoDB shell.
For interactive help, type "help".
For more comprehensive documentation, see
    http://docs.mongodb.org/
Questions? Try the support group
```

MongoDB- Creating a Database

- To create a database in MongoDB, start by creating a MongoClient object, then specify a connection URL with the correct ip address and the name of the database you want to create.
- MongoDB will create the database if it does not exist, and make a connection to it.
- **use DATABASE_NAME** is used to create database in mongo client

```
> use employee_db  
switched to db employee_db  
>
```

- **> show dbs** → See the list of databases
- **> db.dropDatabase()** → delete current database

Creating a Collection

- A **collection** in MongoDB is the same as a **table** in MySQL.
- To create a collection in MongoDB, use the `createCollection()` method.
- MongoDB waits until you have inserted a document before it actually creates the collection.
- In Mongo Client command prompt,
> `db.createCollection("emp1_collection");`
{ "ok" : 1 }
> `show collections;` → *display all collections (tables)*
> `db.emp1_collection.drop()` → *delete a collection* returns true

```
> db.createCollection("emp_collection");
{ "ok" : 0, "errmsg" : "collection already exists", "code" : 48 }
> db.createCollection("emp1_collection");
{ "ok" : 1 }
> show collections;
emp1_collection
emp_collection
>
```

MongoDB supports many datatypes

- **String** – This is the most commonly used datatype to store the data. String in MongoDB must be UTF-8 valid.
- **Integer** – This type is used to store a numerical value. Integer can be 32 bit or 64 bit depending upon your server.
- **Boolean** – This type is used to store a boolean (true/ false) value.
- **Double** – This type is used to store floating point values.
- **Min/ Max keys** – This type is used to compare a value against the lowest and highest BSON elements.
- **Arrays** – This type is used to store arrays or list or multiple values into one key.
- **Timestamp** – 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 Into Collection

- A **document** in MongoDB is the same as a **record** in MySQL.
- To **insert a record**, or *document* as it is called in MongoDB, into a collection, we use the **insertOne()** method.
- The first parameter of the insertOne() method is an object containing the name(s) and value(s) of each field in the document you want to insert.

```
> db.emp_collection.insert({name:"vijayan",school:"SITE"})  
WriteResult({ "nInserted" : 1 })
```

```
> db.emp_collection.insert({name:"vijayan",school:"SITE"});  
WriteResult({ "nInserted" : 1 })  
>
```


- To insert multiple documents in a single query, you can pass an array of documents in **insert()** command

>**db.emp_collection.insert**([{name:"marees",school:"SCOPE"},{name:"eswari",school:"SENSE"}]);

```
> db.emp_collection.insert([{name:"marees",school:"SCOPE"},{name:"eswari",school:"SENSE"}]);
BulkWriteResult<<
  "writeErrors" : [ ],
  "writeConcernErrors" : [ ],
  "nInserted" : 2,
  "nUpserted" : 0,
  "nMatched" : 0,
  "nModified" : 0,
  "nRemoved" : 0,
  "upserted" : [ ]
>>
>
```

- **insertOne() method**

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

```
db.empDetails.insertOne( { First_Name: "Radhika", Last_Name: "Sharma", Date_Of_Birth: "1995-09-26", e_mail: "radhika_sharma.123@gmail.com", phone: "9848022338" })
```

- **insertMany() method**

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

```
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" } ] )
```

Query Document - Display values

- `> db.COLLECTION_NAME.find()` → **find()** method will display all the documents in a non-structured way.

```
> db.emp_collection.find();
{ "_id" : ObjectId("5badbce1b701835b49de0e79"), "name" : "marees", "school" : "S
COPE" }
{ "_id" : ObjectId("5badbce1b701835b49de0e7a"), "name" : "eswari", "school" : "S
ENSE" }
> db.emp_collection.find().pretty()
```

- 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)

To improve the readability, we can format the output in JSON format with this command:

- `db.emp_collection.find().pretty();` (or)
- `db.emp_collection.find().forEach(printjson);`

```
> db.emp_collection.find().forEach(printjson);
{
  "_id" : ObjectId("5badbce1b701835b49de0e79"),
  "name" : "marees",
  "school" : "SCOPE"
}
{
  "_id" : ObjectId("5badbce1b701835b49de0e7a"),
  "name" : "eswari",
  "school" : "SENSE"
}
{
  "_id" : ObjectId("5badc420b701835b49de0e7b"),
  "name" : "marees",
  "school" : "SITE"
}
```

Query Document based on the criteria

- **Equality Criteria:** I want to fetch the data of "marees" from emp_collection. The command for this should be:

```
> db.emp_collection.find({name:"marees"}).pretty();
{
  "_id" : ObjectId("5badbce1b701835b49de0e79"),
  "name" : "marees",
  "school" : "SCOPE"
}
{
  "_id" : ObjectId("5badc420b701835b49de0e7b"),
  "name" : "marees",
  "school" : "SITE"
}
```

- **Greater Than Criteria:**

- I would like to fetch the details of students having age > 32 then the query should be:

- `db.students.find({"age":{"$gt":32}}).pretty()`

- **Less than Criteria:**

- `db.students.find({"StudentId":{"$lt":3000}}).pretty()`

- **Not Equals Criteria:**

- `db.students.find({"StudentId":{"$ne":1002}}).pretty()`

- **Greater than equals Criteria:**

- `db.collection_name.find({"field_name":{"$gte":criteria_value}}).pretty()`

- **Less than equals Criteria:**

- `db.collection_name.find({"field_name":{"$lte":criteria_value}}).pretty()`

- **Equality:**

- `db.students.find({"age":"32"}).pretty()`

- **The findOne() method**

- Apart from the find() method, there is **findOne()** method, that returns only one document.
- `db.mycol.findOne({title: "MongoDB Overview"})`

- **AND,OR ,NOT**

- `db.mycol.find({ $and: [{<key1>:<value1>}, {<key2>:<value2>}] })pretty()`
- `db.mycol.find({ $or: [{key1: value1}, {key2:value2}] }).pretty()`
- `db.mycol.find({ $not: [{key1: value1}, {key2:value2}] })`

Updating Document

- `db.collection_name.update(selection criteria, update_data)`

> **db.emp_collection.update**({school:"SENSE"},
{**\$set**:{school:"SELECT"}})

WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })

```
> db.emp_collection.update({school:"SENSE"},{$set:{school:"SELECT"}});
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.emp_collection.find().pretty();
{
  "_id" : ObjectId("5badbce1b701835b49de0e79"),
  "name" : "marees",
  "school" : "SCOPE"
}
{
  "_id" : ObjectId("5badbce1b701835b49de0e7a"),
  "name" : "eswari",
  "school" : "SELECT"
}
{
  "_id" : ObjectId("5badc420b701835b49de0e7b"),
  "name" : "marees",
  "school" : "SITE"
}
```


- By default the **update method updates a single document**. In the above example we had only one document matching with the criteria, however if there were more then also only one document would have been updated. To enable update() method to **update multiple documents** you have to **set "multi" parameter of this method to true** as shown below.

➤ **db.emp_collection.update({name:"marees"},
{ \$set:{name:"vijayan"} }, { **multi:true** })**

- **WriteResult({ "nMatched" : 2, "nUpserted" : 0,
"nModified" : 2 })**

- **save() Method**

- The **save()** method replaces the existing document with the new document passed in the save() method.
- `db.COLLECTION_NAME.save({_id:ObjectId(),NEW_DATA})`

- **findOneAndUpdate() method**

- The **findOneAndUpdate()** method updates the values in the existing document.
- `db.COLLECTION_NAME.findOneAndUpdate(SELECTIOIN_CRITERIA, UPDATED_DATA)`
- **Eg-** `db.empDetails.findOneAndUpdate({First_Name: 'Radhika'}, { $set: { Age: '30',e_mail: 'radhika_newemail@gmail.com'}})`

- **updateOne() method**

- This method updates a single document which matches the given filter.
- `db.COLLECTION_NAME.updateOne(<filter>, <update>)`
- **Eg-** `db.empDetails.updateOne({First_Name: 'Radhika'}, { $set: { Age: '30', e_mail: 'radhika_newemail@gmail.com' } }`

- **updateMany() method**

- The `updateMany()` method updates all the documents that match the given filter.
- `db.COLLECTION_NAME.updateMany(<filter>, <update>)`
- `db.empDetails.updateMany({Age: { $gt: "25" } }, { $set: { Age: '00' } })`

Delete Document from a Collection

- **db.collection_name.remove(delete_criteria)**

```
> db.emp_collection.remove(<name:"marees">);  
WriteResult(< "nRemoved" : 0 >)  
> db.emp_collection.remove(<name:"mareeswari">);  
WriteResult(< "nRemoved" : 2 >)  
>
```

- When there are more than one documents present in collection that matches the criteria then **all those documents will be deleted** if you run the remove command. However there is a way to limit the deletion to only one document so that even if there are more documents matching the deletion criteria, only one document will be deleted.
- **db.collection_name.remove(delete_criteria, justOne)**
- Here **justOne** is a Boolean parameter that takes only **1 and 0**, if you give 1 then it will limit the document deletion to only 1 document.

Projection – select particulars

- to get the selected fields of the documents rather than all fields.
- Value 1 means show that field and 0 means do not show that field. When we set a field to 1 in Projection other fields are automatically set to 0, except `_id`, so to avoid the `_id` we need to specifically set it to 0 in projection. The vice versa is also true when we set few fields to 0, other fields set to 1 automatically.
- `db.COLLECTION_NAME.find({}, {KEY:1})`

```
> db.emp_collection.find(<>, {name:1});
{ "_id" : ObjectId("5badbce1b701835b49de0e7a"), "name" : "eswari" }
{ "_id" : ObjectId("5bae072bb701835b49de0e7c"), "name" : "marees" }
{ "_id" : ObjectId("5bae074db701835b49de0e7d"), "name" : "marees" }
{ "_id" : ObjectId("5bae0769b701835b49de0e7e"), "name" : "venkat" }
> db.emp_collection.find(<>, {_id:0, name:1});
{ "name" : "eswari" }
{ "name" : "marees" }
{ "name" : "marees" }
{ "name" : "venkat" }
>
```

- **The Limit() Method**

- To limit the records in MongoDB, you need to use **limit()** method. The method accepts one number type argument, which is the number of documents that you want to be displayed.
- `db.COLLECTION_NAME.find().limit(NUMBER)`
- `>db.mycol.find({}, {"title":1, _id:0}).limit(2)`
`{ "title": "MongoDB Overview" } { "title": "NoSQL Overview" } >`

- **Skip() Method**

- **skip()** which also accepts number type argument and is used to skip the number of documents.
- `>db.COLLECTION_NAME.find().limit(NUMBER).skip(NUMBER)`
- `>db.mycol.find({}, {"title":1, _id:0}).limit(1).skip(1) { "title": "NoSQL Overview" } >`
- `{_id : ObjectId("507f191e810c19729de860e1"), title: "MongoDB Overview"},`
- `{_id : ObjectId("507f191e810c19729de860e2"), title: "NoSQL Overview"},`
- `{_id : ObjectId("507f191e810c19729de860e3"), title: "Tutorials Point Overview"}`

- **Sorting Documents using sort() method**
- `db.collection_name.find().sort({field_key:1 or -1})`
- 1 is for ascending order and -1 is for descending order. The default value is 1.

```
> db.emp_collection.find().sort({school:1}).pretty();
{
  "_id" : ObjectId("5bae074db701835b49de0e7d"),
  "name" : "marees",
  "school" : "SCOPE"
}
{
  "_id" : ObjectId("5bae0769b701835b49de0e7e"),
  "name" : "venkat",
  "school" : "SCOPE"
}
{
  "_id" : ObjectId("5badbce1b701835b49de0e7a"),
  "name" : "eswari",
  "school" : "SELECT"
}
{
  "_id" : ObjectId("5bae072bb701835b49de0e7c"),
  "name" : "marees",
  "school" : "SITE"
}
```

- To display the *school* field of all the *emp_collection* in **ascending order** and display it.

```
> db.emp_collection.find(<{}>,{school:1,_id:0}).sort(<{school:1}>).pretty();  
{ "school" : "SCOPE" }  
{ "school" : "SCOPE" }  
{ "school" : "SELECT" }  
{ "school" : "SITE" }
```

- **The default is ascending order**
- > db.emp1_collection.find({}, {school: 1, _id:0}).sort({})

Regular Expression - \$regex

- MongoDB also provides functionality of regular expression for string pattern matching using the **\$regex** operator.
 - `db.posts.find({post_text:{$regex:"tutorialspoint"}}).pretty()`
 - `db.posts.find({post_text:/tutorialspoint/})`
- search case **insensitive**, we use the **\$options** parameter with value **\$i**
- `db.posts.find({post_text:{$regex:"tutorialspoint",$options:"$i"}})`

Aggregation

- Aggregations operations process data records and return computed results. Aggregation operations group values from multiple documents together, and can perform a variety of operations on the grouped data to return a single result.
- `db.COLLECTION_NAME.aggregate(AGGREGATE_OPERATION)`