

**Python online sessions**

**Python --->Oracle DB and MySQL database**

**MongoDB**

**MangoDB or MongoDB???**

**MongoDB**

**Humongous**

**extremely large : huge a humongous building humongous amounts of money.**

**MongoDB is the most popular and trending database.**

**The vendor: MongoDB**

**<https://www.mongodb.com/>**

**Where we can use MongoDB database?**

-----

**We can use everywhere**

**For desktop applications, for mobile applications**

**For web applications, this database is more popular.**

**Full stack developer:**

**stack--->The technologies which can be used to develop web applications are called a stack.**

**The most popular stacks:**

**1. MEAN Stack**

**2. MERN Stack**

## **1. MEAN Stack:**

-----

**M --->MongoDB**

**E --->Express**

**A --->Angular**

**N --->Node JS**

## **2. MERN Stack:**

-----

**M --->MongoDB**

**E --->Express**

**R --->React JS**

**N --->Node JS**

**React or Angular is Front end framework.**

**Express JS is the backend server side framework**

**Node JS is responsible to provide server side runtime environment.**

**MongoDB -->Database**

**By using MEAN and MERN stacks we can build javascript based web applications.**

**Node JS is based on Java Script**

**React or Angular is based on Java Script**

**Express is based on Java Script**

**MongoDB is also based on Java Script.**

**MongoDB internally used Mozilla's Spider Monkey Java Script Engine.**

**Q. What is the type of MongoDB database?**

**It is Document Database/NoSql database.**

## **Relational Database vs Document Database:**

---

**There are 2 most common types of databases.**

**1. Relational Databases/SQL Databases**

**2. Document Databases/NoSQL Databases**

**1. Relational Databases/SQL Databases**

---

**The data will be stored in tables and these tables has fixed schema.  
Employee(eno,ename,esal,eaddr)**

**The data stored in tables has relationships like**

**The data stored in tables has relationships like**

**one to one**

**one to many**

**many to one**

**etc**

**To retrieve data from relational databases, we have to write join queries which collects data from different tables.**

**eg: Oracle,MySQL etc**

**2. Document Databases/NoSQL Databases:**

---

**Data will be stored in separate documents and each document is independent of others.**

**eg: MongoDB**

## **MongoDB Structure:**

-----

**MongoDB Physical database contains several logical databases.**

**Each database contains several collections. Collection is something like table in relational database.**

**Each collection contains several documents. Document is something like record/row in relational database.**

**eg:**

**Database: Shopping cart database**

**Collections: Customers, Products, Orders**

**Cusomer Collection: contains several documents**

**document1:**

```
{  
  "Name":"Sunny",  
  "age":40,  
  "Salary":10000  
}
```

**document2:**

```
{  
  "Name":"Durga"  
}
```

**document-3:**

```
{  
  "name":"Bunny",  
  "age":30,  
  "address":  
    {  
      "city":"Hyderabad"  
    },  
}
```

```
"hobbies":[  
    {"name":"Cricket playing"},  
    {"name":"swimming"}  
]  
}
```

**Q. How data represented in MongoDB?**

**In JSON (BSON) Format.**

**JSON--->Java Script Object Notation**

**BSON--->Binary JSON**

**Key Characteristics of MongoDB database:**

-----

**1. Installation and setup is very easy.**

**1. All information related to a document will be stored in a single place.**

**To retrieve data, it is not required to perform join operations and hence retrieval is very fast.**

**2. Documents are independent of each other and no schema. Hence we can store unstructured data like videos, audio files etc**

**3. We can perform operations like editing existing document, deleting document and inserting new documents very easily.**

**4. Retrieval data is in the form of json which can be understandable by any programming language without any conversion (interoperability)**

**5. We can store very huge amount of data and hence scalability is more.**

**Note: Performance and Flexibility are biggest assets of MongoDB.**

**500 Rs**

**durgasoftonlinetraining@gmail.com**

**8885252627**

**8096969696**

**6 months videos**

**durgasoftonline.com**

**Title-Introduction to MongoDB**

**<https://www.youtube.com/watch?v=wzrwN2wGJwU>**

**MongoDB Shell vs MongoDB Server:**

-----

**Once we installed MongoDB, we will get MongoDB Shell and MongoDB Server.**

**These are Javascript based applications.**

**MongoDB Server is responsible to store our data in database.**

**MongoDB Shell is responsible to manage Server.**

**By using this shell we can perform all required CRUD operations.**

**C --->Create**

**R --->Retrieve**

**U --->Update**

**D --->Delete**

**sir in mongo db all crud operations will be related to documents ?**

**Yes**

**MongoDB Server can be either local or remote.**

**To Launch/Start MongoDB Server --->mongod command**

**To Launch/Start MongoDB Shell --->mongo command**

**GUI Support is also there for MongoDB Shell--->**

**Compass**

**Robo T3**

**etc**

**MongoDB Drivers:**

-----

**From Application(Java,Python,C# etc) if we want to communicate with database, some special software must be required, which is nothing but Driver software.**

**mongodb.com--->Docs-->Drivers**

**<https://pymongo.readthedocs.io/en/stable/tutorial.html>**

**27017**

**wat is difference between oracle DB nd Mango DB**

**if I learns mdb can I work on elastic search**

**Oracle-->MySQL**

**what is the difference between Enterprise and Community versions ??**

**any extra features in enterprise version ??**

**MongoDB Shell, Server and Driver**

**Material**

**Running Notes**

**Videos**

**MongoDB Installation:**

-----

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

**C:\Program Files\MongoDB\Server\4.4\bin>mongod -version**

**db version v4.4.2**

**Build Info: {**

**"version": "4.4.2",**

**"gitVersion": "15e73dc5738d2278b688f8929aee605fe4279b0e",**

**"modules": [],**

**"allocator": "tcmalloc",**

**"environment": {**

**"distmod": "windows",**

**"distarch": "x86\_64",**

**"target\_arch": "x86\_64"**

**}**

**}**

**C:\Program Files\MongoDB\Server\4.4\bin>mongo -version**

**MongoDB shell version v4.4.2**

**Build Info: {**

**"version": "4.4.2",**

**"gitVersion": "15e73dc5738d2278b688f8929aee605fe4279b0e",**

**"modules": [],**

**"allocator": "tcmalloc",**



```
"environment": {  
  "distmod": "windows",  
  "distarch": "x86_64",  
  "target_arch": "x86_64"  
}  
}
```

**{"error": "NonExistentPath: Data directory D:\\data\\db\\ not found.  
Create the missing directory or specify another path using (1) the --  
dbpath command line option, or (2) by adding the 'storage.dbPath'  
option in the configuration file."}}**

**mongod --dbpath "C:\\data\\db"**

**> db.version()**

**4.4.2**

**> db.help()**

**DB methods:**

**db.adminCommand(nameOrDocument) - switches to 'admin' db,  
and runs command [just calls db.runCommand(...)]**

**db.aggregate([pipeline], {options}) - performs a collectionless  
aggregation on this database; returns a cursor**

**db.auth(username, password)**

**db.cloneDatabase(fromhost)** - will only function with MongoDB 4.0 and below

**db.commandHelp(name)** returns the help for the command

**db.copyDatabase(fromdb, todb, fromhost)** - will only function with MongoDB 4.0 and below

**db.createCollection(name, {size: ..., capped: ..., max: ...})**

**db.createUser(userDocument)**

**db.createView(name, viewOn, [{operator: {...}}, ...], {viewOptions})**

**db.currentOp()** displays currently executing operations in the db

**db.dropDatabase(writeConcern)**

**db.dropUser(username)**

**db.eval()** - deprecated

**db.fsyncLock()** flush data to disk and lock server for backups

**db.fsyncUnlock()** unlocks server following a **db.fsyncLock()**

**db.getCollection(cname)** same as **db['cname']** or **db.cname**

**db.getCollectionInfos([filter])** - returns a list that contains the names and options of the db's collections

**db.getCollectionNames()**

**db.getLastError()** - just returns the err msg string

**db.getLastErrorObj()** - return full status object

**db.getLogComponents()**

**db.getMongo()** get the server connection object

**db.getMongo().setSecondaryOk()** allow queries on a replication secondary server

**db.getName()**

**db.getProfilingLevel()** - deprecated

**db.getProfilingStatus()** - returns if profiling is on and slow threshold

**db.getReplicationInfo()**

**db.getSiblingDB(name)** get the db at the same server as this one

**db.getWriteConcern()** - returns the write concern used for any operations on this db, inherited from server object if set

**db.hostInfo()** get details about the server's host

**db.isMaster()** check replica primary status  
**db.hello()** check replica primary status  
**db.killOp(opid)** kills the current operation in the db  
**db.listCommands()** lists all the db commands  
**db.loadServerScripts()** loads all the scripts in db.system.js  
**db.logout()**  
**db.printCollectionStats()**  
**db.printReplicationInfo()**  
**db.printShardingStatus()**  
**db.printSecondaryReplicationInfo()**  
**db.resetError()**  
**db.runCommand(cmdObj)** run a database command. if cmdObj is a string, turns it into {cmdObj: 1}  
**db.serverStatus()**  
**db.setLogLevel(level,<component>)**  
**db.setProfilingLevel(level,slowms)** 0=off 1=slow 2=all  
**db.setVerboseShell(flag)** display extra information in shell output  
**db.setWriteConcern(<write concern doc>)** - sets the write concern for writes to the db  
**db.shutdownServer()**  
**db.stats()**  
**db.unsetWriteConcern(<write concern doc>)** - unsets the write concern for writes to the db  
**db.version()** current version of the server  
**db.watch()** - opens a change stream cursor for a database to report on all changes to its non-system collections.  
**> db.stats()**  
**{**  
     **"db" : "test",**  
     **"collections" : 0,**  
     **"views" : 0,**  
     **"objects" : 0,**  
     **"avgObjSize" : 0,**

```
"dataSize" : 0,  
"storageSize" : 0,  
"totalSize" : 0,  
"indexes" : 0,  
"indexSize" : 0,  
"scaleFactor" : 1,  
"fileSize" : 0,  
"fsUsedSize" : 0,  
"fsTotalSize" : 0,  
"ok" : 1  
}
```

```
> show dbs
```

```
admin 0.000GB
```

```
config 0.000GB
```

```
local 0.000GB
```

```
> use admin
```

```
switched to db admin
```

```
> show collections
```

```
system.version
```

```
> use local
```

```
switched to db local
```

```
> show collections
```

```
startup_log
```

**mongodb--->install-->Physical database**

**estoredb--->**

**collegedb-->**

**data/db everytime**

**500 Rs**

**Running Notes**

**Material PDFs**

**Videos 6 months access (durgasoftonline.com)**

**Colorful Course Completion Certificate**

**15 days to provide basic overview of mongodb.**

**Session-1: <https://www.youtube.com/watch?v=wzrwN2wGJwU>**

**Session-2: <https://www.youtube.com/watch?v=Ncz7h80IzC4>**

**Default Databases:**

-----

**MongoDB Admin will use these default databases.**

**> show dbs**

**admin 0.000GB**

**config 0.000GB**

**local 0.000GB**

**1. admin:**

-----

**admin database is used to store user authentication and authorization information like usernames,passwords,roles etc**

**This database is used by administrators while creating,deleting and updating users and while assigning roles.**

## **2. config:**

-----

**To store configuration information of mongodb server.**

## **3. local:**

-----

**local database can be used by admin while performing replication process.**

## **Data Formats in MongoDB:**

-----

**json: {name:'durga'}--->BSON and that BSON will be stored**

## **BSON: Binary JSON**

**End user/Developer will provide data in json form.**

**In MongoDB server data will be stored in BSON Form.**

## **1. In Javascript only 6 types are available.**

**String,Number,Object,Array,Boolean,Null**

**But BSON provides some extra types also like**

**32-Bit Integer-->NumberInt**

**ObjectId**

**Date**

**etc**

## **2. BSON Format requires less memory.**

**JSON-->10KB**

**BSON--->4 to 5 KB**

**Note: Efficient Storage and Extra data types are speciality of BSON over JSON.**

**While retrieving also, BSON will be converted to JSON by MongoDB server??**

**EJSON--->Extended JSON**

**At the time of retrieval BSON data will be converted to EJSON for understanding purpose.**

**Insertion of Document/Creation --->JSON to BSON**

**Read Operation/Retrieval Operation -->BSON to EJSON**

**Q. What data formats used in MongoDB?**

**3 formats: JSON,BSON,EJSON**

**Creation of Database and Collection:**

-----

**Database won't be created at the beginning and it will be created dynamically.**

**Whenever we are creating collection or inserting document then database will be created dynamically.**

**> show dbs**

```
admin 0.000GB
config 0.000GB
local 0.000GB
> use durgadb
switched to db durgadb
> show dbs
admin 0.000GB
config 0.000GB
local 0.000GB
```

**How to create collection:**

-----

```
db.createCollection("employees")
```

```
> show dbs
admin 0.000GB
config 0.000GB
local 0.000GB
> db.createCollection("employees")
{ "ok" : 1 }
> show dbs
admin 0.000GB
config 0.000GB
durgadb 0.000GB
local 0.000GB
> show collections
employees
```

**Q. But here we said, db.createCollection. How it understood it to create in durgadb?**



**Because we already switched to durgadb because of "use durgadb" command**

**Q. How to drop collection?**

**db.collection.drop()**

**db.students.drop()**

**> show collections**

**employees**

**students**

**> db.students.drop()**

**true**

**> show collections**

**employees**

**so it is mandatory to be in the required DB to drop the collection**

**Q. How to drop database?**

**db.dropDatabase()**

**current database will be deleted.**

**> show dbs**

**admin 0.000GB**

**config 0.000GB**

**durgadb 0.000GB**

**local 0.000GB**

**> db.dropDatabase()**

**{ "dropped" : "durgadb", "ok" : 1 }**

**> show dbs**

**admin 0.000GB**

**config 0.000GB**

**local 0.000GB**

**Sir, from anywhere like from another database can we drop other database??**

**Note: db.getName() to know current database.**

**Basic CRUD Operations in simple way:**

-----

**1. C--->Create|Insert document**

-----

**How to insert document into the collection**

**db.collection.insertOne()**

**db.collection.insertMany()**

**db.collection.insert()**

**db.employees.insertOne({eno:100,ename:"Sunny",esal:1000,eaddr:"Hyd"})**

**2. R--->Read / Retrieval Operation:**

-----

**db.collection.find() --->To get all documents present in the given collection**

**db.collection.findOne() --->To get one document**

**eg: db.employees.find()**

**> db.employees.find()**

**{ "\_id" : ObjectId("5fe16d547789dad6d1278927"), "eno" : 100, "ename" : "Sunny", "esal" : 1000, "eaddr" : "Hyd" }**

```

{ "_id" : ObjectId("5fe16da07789dad6d1278928"), "eno" : 200, "ename"
: "Bunny", "esal" : 2000, "eaddr" : "Mumbai" }
{ "_id" : ObjectId("5fe16dc67789dad6d1278929"), "eno" : 300, "ename"
: "Chinny", "esal" : 3000, "eaddr" : "Chennai" }
{ "_id" : ObjectId("5fe16ddb7789dad6d127892a"), "eno" : 400, "ename"
: "Vinny", "esal" : 4000, "eaddr" : "Delhi" }

```

```

> db.employees.find().pretty()

```

```

{
  "_id" : ObjectId("5fe16d547789dad6d1278927"),
  "eno" : 100,
  "ename" : "Sunny",
  "esal" : 1000,
  "eaddr" : "Hyd"
}
{
  "_id" : ObjectId("5fe16da07789dad6d1278928"),
  "eno" : 200,
  "ename" : "Bunny",
  "esal" : 2000,
  "eaddr" : "Mumbai"
}
{
  "_id" : ObjectId("5fe16dc67789dad6d1278929"),
  "eno" : 300,
  "ename" : "Chinny",
  "esal" : 3000,
  "eaddr" : "Chennai"
}
{
  "_id" : ObjectId("5fe16ddb7789dad6d127892a"),
  "eno" : 400,
  "ename" : "Vinny",

```

```
"esal" : 4000,  
"eaddr" : "Delhi"  
}
```

### **3. U-->Update Operation:**

-----

```
db.collection.updateOne()  
db.collection.updateMany()  
db.collection.replaceOne()
```

**Update Vinny salary as 10000**

```
db.collection.updateOne()  
db.employees.updateOne({ename: "Vinny"},{esal:10000})  
if esal field is available then old value will be replaced with 10000.  
If the field is not already available then it will be created.
```

```
> db.employees.updateOne({ename: "Vinny"},{esal:10000})  
uncaught exception: Error: the update operation document must  
contain atomic operators :  
DBCollection.prototype.updateOne@src/mongo/shell/crud_api.js:565:19  
@(shell):1:1
```

```
db.employees.updateOne({ename: "Vinny"},{$set: {esal:10000}})
```

```
> db.employees.updateOne({ename: "Vinny"},{$set: {esal:10000}})  
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
```

```
> db.employees.updateOne({ename: "Vinny"},{$set: {esal:10000}})  
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }  
> db.employees.find().pretty()
```

```

{
  "_id" : ObjectId("5fe16d547789dad6d1278927"),
  "eno" : 100,
  "ename" : "Sunny",
  "esal" : 1000,
  "eaddr" : "Hyd"
}
{
  "_id" : ObjectId("5fe16da07789dad6d1278928"),
  "eno" : 200,
  "ename" : "Bunny",
  "esal" : 2000,
  "eaddr" : "Mumbai"
}
{
  "_id" : ObjectId("5fe16dc67789dad6d1278929"),
  "eno" : 300,
  "ename" : "Chinny",
  "esal" : 3000,
  "eaddr" : "Chennai"
}
{
  "_id" : ObjectId("5fe16ddb7789dad6d127892a"),
  "eno" : 400,
  "ename" : "Vinny",
  "esal" : 10000,
  "eaddr" : "Delhi"
}

```

**Note:** If anything prefixed with \$ symbol, then it is predefined word in MongoDB.

#### **4. D -->Delete:**

-----

```
db.collection.deleteOne()  
db.collection.deleteMany()
```

```
db.employees.deleteOne({ename:"Vinny"})
```

**Note: database and collection will be created dynamically whenever we are trying to insert documents.**

```
> use studentdb  
switched to db studentdb  
> db.students.insertOne({name:"Durga",rollno:101,marks:98})  
{  
    "acknowledged" : true,  
    "insertedId" : ObjectId("5fe172617789dad6d127892b")  
}  
> show dbs  
admin      0.000GB  
config     0.000GB  
durgadb    0.000GB  
local      0.000GB  
studentdb  0.000GB  
> show collections  
students  
> db.students.find().pretty()  
{  
    "_id" : ObjectId("5fe172617789dad6d127892b"),  
    "name" : "Durga",  
    "rollno" : 101,  
    "marks" : 98  
}
```

**Session-1: <https://www.youtube.com/watch?v=wzrwN2wGJwU>**

**Session-2: <https://www.youtube.com/watch?v=Ncz7h80IzC4>**

## **CRUD**

### **Capped Collections:**

-----

**> use durgadb**

**> db.createCollection("employees")**

**db.createCollection(name)**

**db.createCollection(name,options)**

**capped**

**max 1000 documents--->1001 document**

**size: 3736578 bytes only-->if space completed**

**db.createCollection("employees",{capped: true, size: 3736578, max: 1000})**

**old documents will be deleted automatically.**

**on what basis old document will be deleted?**

**based on timestamp**

**oldest document will be deleted automatically.**

**If capped is true means that if size exceeds or maximum number of documents reached, then oldest entry will be deleted automatically.**

**1. db.createCollection("employees") --->Normal Collection**

**2. db.createCollection("employees",{capped: true})--->Invalid**

**"errmsg" : "the 'size' field is required when 'capped' is true",**

**3. db.createCollection("employees",{capped: true, size: 365675})--->valid**

**4. db.createCollection("employees",{size: 365675})--->invalid**  
**"errmsg" : "the 'capped' field needs to be true when either the 'size' or 'max' fields are present"**

**5. db.createCollection("employees",{capped: true, size: 365675, max: 1000})--->valid**

**6. db.createCollection("employees",{capped: true, size: 365675, max: 1})**

**use case:**

-----

**freshers jobs portal--->students registered**  
**students collection---> 1 lakh**

**jobs collection---> jobs 100 jobs**

**Q. What is capped collection?**

**If size exceeds or maximum number of documents reached, then oldest entry will be deleted automatically, such type of collections are called capped collections.**

## **CRUD**

**Inserting Documents in the collection(C--->Create):**

-----

**db.collection.insertOne()**  
**db.collection.insertMany()**  
**db.collection.insert()**

**db.collection.insertOne():**



-----  
**To insert only one document.**

**db.employees.insertOne({...})**

**Argument is only one javascript object.**

**db.employees.insertOne({eno: 100, ename: "Sunny", esal: 1000, eaddr: "Mumbai"})**

**db.collection.insertMany():**

-----  
**To insert multiple documents**

**db.collection.insertMany([{{..}},{{..}},{{..}},{{..}}])**

**db.employees.insertMany([{{eno: 200, ename: "Sunny", esal: 1000, eaddr: "Mumbai"}},{{eno: 300, ename: "Sunny", esal: 1000, eaddr: "Mumbai"}}])**

**db.collection.insert():**

-----  
**To insert either a single document or multiple documents.**

**db.employees.insert({...})**

**db.employees.insert([{{..}},{{..}},{{..}},{{..}}])**

```
db.employees.insert({eno: 700, ename: "Sunny", esal: 1000, eaddr: "Mumbai"})
```

```
db.employees.insert([{eno: 800, ename: "Sunny", esal: 1000, eaddr: "Mumbai"},{eno: 900, ename: "Sunny", esal: 1000, eaddr: "Mumbai"}])
```

**Creating Document separately and inserting into collection:**

---

```
var emp = {};  
emp.eno = 7777;  
emp.ename = "Bunny";  
emp.esal = 777777;  
emp.eaddr = "Hyderabad";
```

```
db.employees.insertOne(emp)  
db.employees.insertMany([emp])  
db.employees.insert(emp)  
db.employees.insert([emp])
```

**Will it make duplicate documents if we add same document multiple times in a collection?**

**\_id**

```
db.employees.insert({_id: "AAA-BBB", name: "Durga"})  
db.employees.insertOne({_id: "AAA-BBB", name: "Ravi"})
```

**What is capped collection?**

## **insertOne() vs insertMany() vs insert()**

### **How to create document before insertion**

**[bit.ly/durgamongodb](http://bit.ly/durgamongodb)**

### **Inserting Documents from java script file:**

-----  
**studentsdb --->database**

**students--->collection**

**in this collection we have to insert documents.**

**students.js:**

-----  
**db.students.insertOne({name: "Durga", rollno: 101, marks: 98 })**  
**db.students.insertOne({name: "Ravi", rollno: 102, marks: 99 })**  
**db.students.insertOne({name: "Shiva", rollno: 103, marks: 100 })**  
**db.students.insertOne({name: "Pavan", rollno: 104, marks: 80 })**

**load("D:\students.js")**

**> show collections**

**> load("D:\students.js")**

**true**

**> show collections**

**students**

**> db.students.find().pretty()**

**{**  
 **"\_id" : ObjectId("5fe40341941e89a2bcd9f34b"),**  
 **"name" : "Durga",**

```

    "rollno" : 101,
    "marks" : 98
  }
  {
    "_id" : ObjectId("5fe40341941e89a2bcd9f34c"),
    "name" : "Ravi",
    "rollno" : 102,
    "marks" : 99
  }
  {
    "_id" : ObjectId("5fe40341941e89a2bcd9f34d"),
    "name" : "Shiva",
    "rollno" : 103,
    "marks" : 100
  }
  {
    "_id" : ObjectId("5fe40341941e89a2bcd9f34e"),
    "name" : "Pavan",
    "rollno" : 104,
    "marks" : 80
  }
}

```

**Q. Does the name of the javascript file has to be the same as collection name??**

**Not required.**

**Inserting Documents from json file (mongoimport tool):**

-----

**In json file, the data should be in array form.  
Make sure the data should be of json only.**

**json vs javascript object:**

-----

**In javascript object, quote symbols for keys are optional.  
But in JSON, quote symbols are mandatory for keys.**

**db.collection.insertOne(javascript object)**

**Here quote symbols are optional**

**students.json:**

-----

```
[
  {
    "name": "Sunny",
    "rollno": 666
  },
  {
    "name": "Bunny",
    "rollno": 777
  },
  {
    "name": "Chinny",
    "rollno": 888
  },
  {
    "name": "Vinny",
    "rollno": 999
  },
  {
    "name": "Pinny",
    "rollno": 555
  }
]
```

**mongod --->tool to start MongoDB Server**

**mongo --->tool to start MongoDB Shell**

**mongoimport --->tool to import documents from json file into MongoDB**

**mongoimport is not available by default. We have to make it available manually.**

**<https://www.mongodb.com/try/download/database-tools>**

**copy mongoimport.exe to the MongoDB bin folder**

**C:\Program Files\MongoDB\Server\4.4\bin**

**\*\*\*\***

**Note: mongoimport command should be executed from the command prompt but not from the shell.**

**Insert all documents from json file into MongoDB**

**database name: rahuldb**

**collection name: students9**

**from the command prompt, go to location where json file is available.**

**mongoimport --db databaseName --collection collectionName --file  
fileName --jsonArray**

**mongoimport --db rahuldb --collection students9 --file students.json --  
jsonArray**

**D:\>mongoimport --db rahuldb --collection students9 --file students.json  
--jsonArray**

**2020-12-24T08:57:34.007+0530 connected to: mongodb://localhost/  
2020-12-24T08:57:34.191+0530 5 document(s) imported successfully.  
0 document(s) failed to import.**

**mongoimport creates database and collection automatically if not available.**

**If collection already available then the new documents will be appended.**

**server must be in running state while we are using mongodimport?  
Server must be in running state.**

**> show dbs**

```
admin      0.000GB
config     0.000GB
dstudentsdb 0.000GB
durgadb    0.000GB
durgadb1   0.000GB
durgadb2   0.000GB
local      0.000GB
rahuldb    0.000GB
storedb    0.000GB
studentdb  0.000GB
```

**> use rahuldb**

**switched to db rahuldb**

**> show collections**

**students9**

**> db.students9.find().pretty()**

```
{
  "_id" : ObjectId("5fe40aa643e59978520a102b"),
  "name" : "Vinny",
  "rollno" : 999
}
{
  "_id" : ObjectId("5fe40aa643e59978520a102c"),
  "name" : "Bunny",
  "rollno" : 777
}
```

```

}
{
  "_id" : ObjectId("5fe40aa643e59978520a102d"),
  "name" : "Chinny",
  "rollno" : 888
}
{
  "_id" : ObjectId("5fe40aa643e59978520a102e"),
  "name" : "Pinny",
  "rollno" : 555
}
{
  "_id" : ObjectId("5fe40aa643e59978520a102f"),
  "name" : "Sunny",
  "rollno" : 666
}

```

**students.json:**

```

-----
[
  {
    "name": "Dhoni",
    "rollno": 7777777
  }
]

```

**mongoimport --db rahuldb --collection students9 --file students.json --  
jsonArray**

**This item it will perform append operation.**



**Note: Similarly, we can use mongoimport to import data from excel/csv files.**

**can i import from anyother RDBMS like oracle? Yes**

**bit.ly/durgamongodb**

**sir please with python can you show the mysql to mongo program**

**How to insert documents from js file**

**How to insert documents from json file**

**How to insert documents from csv file| excel file**

**Nested Documents:**

-----

**Sometimes we can take a document inside another document, such type of documents are called nested documents or embedded documents.**

**employees:**

```
{  
  eno:100,  
  ename:"durga",  
  esal:1000,  
  eaddr:"Hyderabad",  
  hobbies: {h1:"Swimming",h2:"Reading"}  
}
```

**storedb-->database name**

**books-->collection**

**books.json**

-----

```
[
  {
    "title": "Python In Simple Way",
    "isbn": 12345,
    "downloadable": true,
    "no_of_reviews": 10,
    "author": {
      "name": "Daniel Kohen",
      "callname": "Dan"
    }
  },
  {
    "title": "MongoDB In Simple Way",
    "isbn": 45678,
    "downloadable": false,
    "no_of_reviews": 5,
    "author": {
      "name": "Shiva Ramachandran",
      "callname": "Shiva"
    }
  }
]
```

**mongoimport --db storedb --collection books --file books.json --  
jsonArray**

```
[
```

```

{
  "title": "Linux In Simple Way",
  "isbn": 778899,
  "downloadable": true,
  "no_of_reviews":0,
  "author": {
    "name": "Shiva Ramachandran",
    "callname": "Shiva",
    "profile": {
      "exp":8,
      "courses":3,
      "books":2
    }
  }
}
]

```

**Note:**

**Inside Nested document, we can take another document also.  
MongoDB supports upto 100 levels of nesting.**

**sir im confusing with server shell cmd which commads where we need use**

**1. command name and purpose**

**db.collection.find().pretty()**

**db.collection.insertOne()**

**db.collection.insertMany()**

**db.collection.insert()**

**db.collection.updateOne({},{})**

## **Arrays in Documents:**

-----

**Any collection of items is called an array.**

**The items can be strings or numbers or objects.**

**A document can contain arrays also.**

**books.json:**

-----

```
[  
  
  {  
    "title": "Devops In Simple Way",  
    "isbn": 112233,  
    "downloadable": false,  
    "no_of_reviews":20,  
    "tags":["jenkins","git","CI/CD"],  
    "languages":["english","hindi","telugu"],  
    "author": {  
      "name": "Martin Kohenova",  
      "callname": "Mart",  
      "profile": {  
        "exp":8,  
        "courses":3,  
        "books":2  
      }  
    }  
  }  
]
```

```
mongoimport --db storedb --collection books --file books.json --  
jsonArray
```

**sir we use find() to fetch all documents but how to fetch a particular field in the document ?**

**Basic idea about CRUD Operations.**

**C--->Create Operation|Insert Operation**

**ObjectId:**

**-----**

**For every document, MongoDB Server will associate a unique id, which is nothing but ObjectId.**

**It is something like primary key in relational databases.**

**The ObjectId will be assigned to \_id field.**

**"\_id" : ObjectId("5fe6ad34b195d71b16a713c8")**

**ObjectId is not json type and it is of BSON type.**

**ObjectId is of 12 bytes.**

- 1. The first 4 bytes represents the timestamp when this document was inserted.**
- 2. The next 3 bytes represents machine identifier( host name)**
- 3. The next 2 bytes represents process id.**
- 4. The last 3 bytes represents some random increment value.**

**Why this lengthy ObjectId:**

**-----**

**The only reason is uniqueness.**

**mobile number contains 10 digits**

**why 10 digits, just only one digit is enough???**

**To generate timestamp from ObjectId:**

-----

**db.employees.find() --->List out all documents of employees collection**

**db.employees.find()[0] --->List out only first document of employees collection**

**db.employees.find()[0].\_id --->ObjectId of first document**

**db.employees.find()[0].\_id.getTimestamp() --->ObjectId of first document**

**> db.employees.find()[0]**

```
{  
  "_id" : ObjectId("5fe2b6fc9d0c84a052cb9745"),  
  "eno" : 100,  
  "ename" : "Sunny",  
  "esal" : 1000,  
  "eaddr" : "Mumbai"  
}
```

**> db.employees.find()[0].\_id**

**ObjectId("5fe2b6fc9d0c84a052cb9745")**

**> db.employees.find()[0].\_id.getTimestamp()**

**ISODate("2020-12-23T03:18:20Z")**

**{ "\_id" : ObjectId("5fe6b3218c25aae60be989c0"), "A" : 100, "B" : 200 }**

**> db.employees.find({"B":200})**

**{ "\_id" : ObjectId("5fe6b3218c25aae60be989c0"), "A" : 100, "B" : 200 }**

**> db.employees.find({"B":200}).\_id**

**> db.employees.find({"B":200})[0]**

**{ "\_id" : ObjectId("5fe6b3218c25aae60be989c0"), "A" : 100, "B" : 200 }**

**> db.employees.find({"B":200})[0].\_id**

```
ObjectId("5fe6b3218c25aae60be989c0")  
> db.employees.find({"B":200})[0]._id.getTimestamp()  
ISODate("2020-12-26T03:50:57Z")
```

**By using \_id field, we can provide our own value as ObjectId. MongoDB server will generate default ObjectId iff we are not providing any \_id field value.**

**If we provide our own value, it may not provide timestamp, machine identifier, process id etc. Hence it is not recommended to provide our own id.**

**Is it possible to have same \_id for 2 documents?**

**Duplicate ObjectIds possible?**

**No chance at all, even if we provide value explicitly also.**

```
db.employees.insertOne({_id:789, name:"Rahul"})  
db.employees.insertOne({_id:789, name:"Viraj"})
```

```
"errmsg" : "E11000 duplicate key error collection: durgadb.employees  
index: _id_ dup key: { _id: 789.0 }",
```

**ObjectIds are Immutable, ie once we creates/assigns ObjectId we cannot change its value.**

```
{ "_id" : 789, "name" : "Rahul" }  
db.employees.updateOne({_id: 789},{ $set: {_id:9999}})
```

**"errmsg" : "Performing an update on the path '\_id' would modify the immutable field '\_id'",**

**Q. We can use the same ObjectId is for other collection right?**

**Yes possible. Uniqueness is per collection not per database.**

**Q. \_id: 100 and \_id: "100" possible in same collection?**

**Yes because data types are different.**

**Q. Which of the following are TRUE?**

- A) We cannot store documents in collection without ObjectId.**
- B) \_id field will be added automatically by MongoDB, if we are not providing that field explicitly.**
- C) ObjectId is not JSON type and it is of BSON Type.**
- D) Default ObjectId generated by MongoDB is of 12 bytes.**
- E) ObjectIds are unique.**
- F) ObjectIds are Immutable.**
- G) We cannot modify the value of ObjectId after creation.**
- H) Default ObjectId consists of timestamp,machine identifier, processid etc**
- I) The advantage of using default ObjectId is we can get several details like timestamp etc**
- J) If we provide our own ObjectId value, it may not generate timestamp,machine identifier,process id etc.**
- K) All of these.**

**Ans: K**

**Sir I have a query. When I get stuck where to refer? You said you learnt now only. Can u guide us?**



**How to Learn New Technologies very Easily?**

**<https://www.youtube.com/watch?v=AQRdnqVh2NA>**

**[youtube.com/durgasoftware](https://www.youtube.com/durgasoftware)**

**[bit.ly/durgamongodb](https://bit.ly/durgamongodb)**

**[durgasoftonline.com](https://durgasoftonline.com)**

**durgasoftonline materials and videos with red color logo**

**insertOne()**

**insertMany()**

**insert()**

**Inserting documents from javascript file by using load() function**

**Inserting documents from json file by using mongoimport tool**

**Nested Documents**

**Arrays In Documents**

**ObjectId**

**Ordered Insertion**

**WriteConcern**

**Atomiticity**

## **Ordered Insertion in Bulk inserts:**

---

**We can perform bulk inserts either by using insertMany() or insert() methods.**

**All documents present inside given array will be inserted into collection.**

**durgadb database**

**alphabets collection**

**some documents-->bulk insert**

**insert([{} , {} , {} , {} , {}])**

**db.alphabets.insertMany([{"A":"Apple"}, {"B":"Banana"}, {"C":"Cat"}])**

**[{"A":"Apple"}, {"B":"Banana"}, {"C":"Cat"}] ==> Array of Javascript objects**

**> db.alphabets.find()**

**{ "\_id" : ObjectId("5fe7f8998a9854ae87538e18"), "A" : "Apple" }**

**{ "\_id" : ObjectId("5fe7f8998a9854ae87538e19"), "B" : "Banana" }**

**{ "\_id" : ObjectId("5fe7f8998a9854ae87538e1a"), "C" : "Cat" }**

## **Default Behaviour of bulk inserts:**

---

**While performing bulk insert operation, if any document insertion fails then rest of the documents won't be inserted. i.e in bulk inserts, order is important.**

**The documents which are already inserted won't be rolled back.**

```
db.cars.insertMany([{_id: 100, M:"Maruti"},{_id: 100, A:"Audi"},{_id: 300,  
B:"Benz"}])
```

**0**

**1**

**2**

```
db.cars.find().pretty()
```

```
> db.cars.insertMany([{_id: 100, M:"Maruti"},{_id: 100, A:"Audi"},{_id:  
300, B:"Benz"}])
```

```
uncaught exception: BulkWriteError({
```

```
  "writeErrors" : [
```

```
    {
```

```
      "index" : 1,
```

```
      "code" : 11000,
```

```
      "errmsg" : "E11000 duplicate key error collection:
```

```
durgadb.cars index: _id_ dup key: { _id: 100.0 }",
```

```
      "op" : {
```

```
        "_id" : 100,
```

```
        "A" : "Audi"
```

```
      }
```

```
    }
```

```
  ],
```

```
  "writeConcernErrors" : [ ],
```

```
  "nInserted" : 1,
```

```
  "nUpserted" : 0,
```

```
  "nMatched" : 0,
```

```
  "nModified" : 0,
```

```

        "nRemoved" : 0,
        "upserted" : [ ]
    }) :
BulkWriteError({
    "writeErrors" : [
        {
            "index" : 1,
            "code" : 11000,
            "errmsg" : "E11000 duplicate key error collection:
durgadb.cars index: _id_ dup key: { _id: 100.0 }",
            "op" : {
                "_id" : 100,
                "A" : "Audi"
            }
        }
    ],
    "writeConcernErrors" : [ ],
    "nInserted" : 1,
    "nUpserted" : 0,
    "nMatched" : 0,
    "nModified" : 0,
    "nRemoved" : 0,
    "upserted" : [ ]
})
BulkWriteError@src/mongo/shell/bulk_api.js:367:48
BulkWriteResult/this.toError@src/mongo/shell/bulk_api.js:332:24
Bulk/this.execute@src/mongo/shell/bulk_api.js:1186:23
DBCollection.prototype.insertMany@src/mongo/shell/crud_api.js:326:5
@(shell):1:1

> db.cars.find().pretty()
{ "_id" : 100, "M" : "Maruti" }

```

**While performing bulk insert operation, if any document insertion fails then rest of the documents won't be inserted. i.e in bulk inserts, order is important.**

**The documents which are already inserted won't be rolled back.**

**We can customize this behaviour. We can customize in such a way that if one document insertion fails, still the remaining documents can be inserted.**

**For this we have to use ordered property.**

**> db.cars.insertMany([{}},{},...{}],{ordered: false})**

**The default value for ordered is true.**

**if order is false==>still one document insertion fails, rest of the documents will be inserted without any problem.**

**db.cars.insert([{\_id:200,I: "Innova"},{\_id:200,R: "Ritz"},{\_id:300,G: "Gitz"}],{ordered: false})**

**How many documents will be inserted in the collection:**

**Ans: 2**

**Q. How to rollback already inserted documents in the case of any error in bulk inserts?**

**Ans: By using transactions**

**transaction: Either all or None**

**Transfer 10k from my account to sunny account**

**operation-1: debit 10k from my account**

**operation-2: credit 10k to sunny account**

**Q. What is the purpose of ordered property in insert operation?**

**Q. While performing bulk insert operation by using either insertMany() or insert() method, if one document insertion fails, is rest of the documents will be inserted or not?**

**By default: No**

**But we can customize this behaviour by using ordered property.**

**Q1. Assume cars collection is empty.**

**db.cars.insert([{\_id:200,I: "Innova"},{\_id:200,R: "Ritz"},{\_id:300,G: "Gitz"}],{ordered: true})**

**How many records will be inserted in the collection?**

**Ans: 1**

**Q2. Assume cars collection is empty.**

**db.cars.insert([{\_id:200,I: "Innova"},{\_id:200,R: "Ritz"},{\_id:300,G: "Gitz"}])**

**How many records will be inserted in the collection?**

**Ans: 1**

**Q3. Assume cars collection is empty.**

**db.cars.insert([{\_id:200,I: "Innova"},{\_id:200,R: "Ritz"},{\_id:300,G: "Gitz"}],{ordered: false})**

**How many records will be inserted in the collection?**

**Ans: 2**

**WriteConcern Property:**

-----

**usecase-1:**

-----

**Whenever we are performing insert operation, by default the shell/client will wait until getting acknowledgement. Server will provide acknowledgement after completing insert operation. This may reduce performance at client side.**

```
> db.cars.insertOne({l:"Innova"})  
{  
  "acknowledged" : true,  
  "insertedId" : ObjectId("5fe800f68a9854ae87538e1b")  
}
```

**We can customize this behaviour by using writeConcern property.**

```
db.collection.insertOne({}, {writeConcern: {w:0}})
```

**w:1==>It is the default value and client will wait until getting acknowledgment.**

**w:0==>It means client won't wait for acknowledgement.**

```
> db.cars.insertOne({B:"BMW"}, {writeConcern: {w: 1} })  
{  
  "acknowledged" : true,  
  "insertedId" : ObjectId("5fe802f48a9854ae87538e1c")  
}  
  
> db.cars.insertOne({H:"Honda"}, {writeConcern: {w: 0} })  
{ "acknowledged" : false }
```

**Even "acknowledged" : false, still document inserted.**

```
> db.cars.find()
{ "_id" : ObjectId("5fe800f68a9854ae87538e1b"), "I" : "Innova" }
{ "_id" : ObjectId("5fe802f48a9854ae87538e1c"), "B" : "BMW" }
{ "_id" : ObjectId("5fe803248a9854ae87538e1d"), "H" : "Honda" }
```

**If lakhs of records are required to insert, if one or two document insertion fails still no problem, but performance is important then writeConcern is the best choice.**

**usecase-2:**

**-----**

**In Production , for every database we have to maintain cloned/replica database because**

- 1. To handle Fail over situations**
- 2. For Load Balancing Purposes**

**A single document is required to insert in multiple database instances like primary database, replica-1, replica-2 etc.**

**Diagram**

**After inserting how many instance, you are expecting acknowledgement, we can specify this by using writeConcern property.**

**if w: 0 ==> No acknowledgement.**

**if w: 1 ==> Acknowledgement after inserting document in primary database.**



**if w: 2 ==> Acknowledgement after inserting document in primary database and replica-1.**

**if w: 3 ==> Acknowledgement after inserting document in primary database, replica-1 and replica-2.**

```
db.cars.insertOne({A:"Audi"},{writeConcern: {w: 3} })  
> db.cars.insertOne({A:"Audi"},{writeConcern: {w: 3} })  
uncaught exception: WriteCommandError({  
  "ok" : 0,  
  "errmsg" : "cannot use 'w' > 1 when a host is not replicated",  
  "code" : 2,  
  "codeName" : "BadValue"  
})
```

**Note: To use 'w' > 1, replica copies should be available already.**

**Note: writeConcern is applicable for any write operation like insert,update and delete.**

**Using mongoimport — db OpenFlights — collection Airport — type csv  
— headerline — ignoreBlanks — file [local path]**

**Importing csv /excel file to mongoDB**

**1. A csv file with following data is getting imported**

**By using mongoimport --db myDb --collection myCollection --type csv --  
headerline --file emp.csv  
eno,ename,esal,eadd  
17325,rkg,100000,lucknow  
gkr,17325,1000,lko**

**12345,yvan,15000,noida**

**2. An excel file with following data when converted to csv by changing Extension is not being imported by above command.**

**sl no name Emp number salary years of service desgination**

**1 AAAAA 12345 10000 4.00 worker**

**2 BBBBB 12346 15000 8.00 worker**

**3 CCCCC 12347 20000 5.00 Manager**

**4 DDDDD 12348 25000 3.00 SM**

**The Big story of insert operation(C->Create Operation)**

-----

- 1. insertOne(),insertMany(),insert()**
- 2. Insert documents from javascript file by using load()**
- 3. Insert documents from json file by using mongoimport**
- 3. Insert documents from csv file by using mongoimport**
- 4. Nested Documents**
- 5. Arrays in Documents**
- 6. ObjectId**
- 7. Ordered Insertion**
- 8. WriteConcern**
- 9. Atomicity**

**Bigger Doubt about Atomicity:**

-----

**Q. Assume we have to insert a document where 100 fields are available, after inserting 50 fields if database server faces some problem then what will be happend?**

**Ans: Whatever fields already added will be rollbacked.**

**MongoDB Server stores either complete document or nothing. ie it won't store part of the document. ie CRUD operations are atomic at document level.**

**db.collection.insertMany([{} ,{} ,{} ,{}])**

**But while inserting multiple documents (Bulk Insertion), after inserting some documents if database server faces some problem, then already inserted documents won't be rolled back. i.e atomicity by default not applicable for bulk inserts.**

**If we want atomicity for bulk inserts then we should go for transactions concept.**

**Transaction: Either All operations or None**

**eg:**

**transfer 10k from my account to sunny account**

**operation-1: debit 10k from my account**

**operation-2: credit 10k to sunny account**

**CRUD Operations-->C**

**CRUD Operations--->R Operation/Read Operation/Retrieve Operation/Find Operation:**

-----

**We can read documents from the collection by using the following find methods.**

- 1. find({query}) --->Returns all matched documents based on query.**
- 2. findOne({query}) --->Returns one matched document based on query.**

**The argument, query is a simple javascript object.**

**These methods are related to collection and hence we have to call these methods on collection object.**

**db.collection.find()  
db.collection.findOne()**

**These find methods are similar to select query in relational databases.**

**eg:**

**read all employees**

**read all employees where esal > 10000**

**read all employees where eaddr is Hyderabad**

**read all employees where eaddr is Hyderabad or esal > 10000**

**read all employees where eaddr is Hyderabad and esal > 10000**

**aggregate functions**

**logical operations**

**etc**

**All these things possible in MongoDB.**

**storedb --->database name**

**books --->collection name**

**books.json:**

**-----**

```
[  
  {  
    "title": "Linux in simple way",  
    "isbn": 6677,  
    "downloadable": false,  
    "no_of_reviews": 1,  
    "tags": ["os","freeware","shell programming"],
```

```

"languages": ["english","hindi","telugu"],
"author": {
  "name": "Shiva Ramachandran",
  "callname": "Shiv",
  "profile": {
    "exp":8,
    "courses":3,
    "books":2
  }
},
{
  "title": "Java in simple way",
  "isbn": 1122,
  "downloadable": true,
  "no_of_reviews": 2,
  "tags": ["language","freeware","programming"],
  "languages": ["english","hindi","telugu"],
  "author": {
    "name": "Karthik Ramachandran",
    "callname": "Karthik",
    "profile": {
      "exp":1,
      "courses":2,
      "books":3
    }
  },
{
  "title": "Python in simple way",
  "isbn": 1234,
  "downloadable": false,
  "no_of_reviews": 5,

```

```

    "tags": ["language","freeware","programming"],
    "languages": ["english","hindi","telugu"],
    "author": {
      "name": "Daniel IA Cohen",
      "callname": "Dan",
      "profile": {
        "exp":8,
        "courses":7,
        "books":6
      }
    }
  },
  {
    "title": "Devops in simple way",
    "isbn": 6677,
    "downloadable": false,
    "no_of_reviews": 3,
    "tags": ["jenkins","git","cicd"],
    "languages": ["english","hindi","telugu"],
    "author": {
      "name": "Dhoni Chandra",
      "callname": "Dhoni",
      "profile": {
        "exp":4,
        "courses":4,
        "books":4
      }
    }
  },
  {
    "title": "MongoDB in simple way",
    "isbn": 6677,
    "downloadable": true,

```

```

    "no_of_reviews": 4,
    "tags": ["database","cloud","nosql"],
    "languages": ["english","hindi","telugu"],
    "author": {
        "name": "Sachin Tendulkar",
        "callname": "Sachin",
        "profile": {
            "exp":6,
            "courses":7,
            "books":8
        }
    },
    {
        "title": "Oracle in simple way",
        "isbn": 6677,
        "downloadable": true,
        "no_of_reviews": 3,
        "tags": ["database","sql","relational"],
        "languages": ["english","hindi","telugu"],
        "author": {
            "name": "Virat Kohli",
            "callname": "kohli",
            "profile": {
                "exp":2,
                "courses":2,
                "books":2
            }
        },
        {
            "title": "Shell Scripting in simple way",
            "isbn": 9988,

```

```

    "downloadable": true,
    "no_of_reviews": 1,
    "tags": ["programming"],
    "languages": ["english","hindi","tamil"],
    "author": {
      "name": "Rama Ramachandran",
      "callname": "Rama",
      "profile": {
        "exp":8,
        "courses":3,
        "books":2
      }
    }
  }
]

```

```

mongoimport --db storedb --collection books --file books.json --
jsonArray

```

**Q1. List out all documents present in books collection?**

```

> db.books.find().pretty()
> db.books.find({}).pretty()

```

**Q2. Find total number of documents present in books collection?**

```

> db.books.find().count()

```

**Q3. List out first document present in books collection?**

```

> db.books.findOne()
> db.books.findOne({})

```

**Note: pretty() and count() methods can be used on find() result but not on findOne() result.**



```
> db.books.findOne().pretty()
```

**uncaught exception: TypeError: db.books.findOne(...).pretty is not a function :**

```
@(shell):1:1
```

```
> db.books.findOne().count()
```

**uncaught exception: TypeError: db.books.findOne(...).count is not a function :**

**Q4. List out all documents from books collection where "downloadable" is false?**

```
> db.books.find({downloadable: false}).pretty()
```

**Q5. List out all documents from books collection where no\_of\_reviews is 3.**

```
> db.books.find({no_of_reviews: 3}).pretty()
```

**Querying Nested Documents:**

-----

**If the value of a field is again a document, then that nested property value can be accessed by using dot operator. In this case, key must be enclosed within quotes.**

**Q1. List out all documents from books collection where author's call name is kohli?**

```
> db.books.find({author.callname: "kohli"}).pretty() ==>invalid
```

```
> db.books.find({"author.callname": "kohli"}).pretty() ==>valid
```

**Q2. List out all documents from books collection where author's profile contains exactly 2 courses?**

**> db.books.find({"author.profile.courses": 2}).pretty()**

### **Querying Array elements:**

-----

**It is exactly same as exact match**

**Q1. List out all documents where 'tags' array contains 'programming' element?**

**> db.books.find({tags: "programming"}).pretty() ==>Query array elements**

**Q2. List out all documents where 'languages' array contains 'telugu' element?**

**> db.books.find({languages: "telugu"}).pretty() ==>Query array elements**

### **Querying Array itself:**

-----

**Q1. List out all documents where 'tags' array contains only one element 'programming'?**

**> db.books.find({tags: ["programming"]}).pretty() ==>Query array itself**

**Q2. List out all documents where 'tags' array is:  
["language","freeware","programming"]**

**> db.books.find({tags: ["language","freeware","programming"]}).pretty()  
Here order of elements inside array is important and even case also.**

**db.books.find({tags: ["language","programming","freeware"]}).pretty()**  
It won't return any document because the given order not matched with any document.

**Note:**

**> db.books.find({tags: "programming"}).pretty() ==>Query array elements**

**> db.books.find({tags: ["programming"]}).pretty() ==>Query array itself**

It is something like we are finding fruit in a basket not basket but in the second case we are trying to find basket itself.

**Query Operators:**

-----

**We can use operators in our queries.**

**Every operator prefixed with \$ symbol to indicate that it is operator but not field or value. By seeing \$ symbol, MongoDB server will execute the corresponding operator functionality.**

**1. Comparison Query Operators:**

-----

**\$eq, \$ne, \$gt, \$gte, \$lt, \$lte, \$in, \$nin**

**\$eq ---> Equality:**

-----

**The \$eq operator matches documents where the value of the field is equal to specified value.**

**Syntax: db.collection.find({ field: {\$eq: value} })**

**It is exactly same as**

**db.collection.find({field: value}) ==>It is the short cut way**

**Case-1: Equals to Specific Value:**

-----

**Q1. Select all documents from books collection, where no\_of\_reviews is 3.**

**> db.books.find({ no\_of\_reviews: { \$eq: 3}}).pretty()**

**> db.books.find({ no\_of\_reviews: 3}).pretty()**

**Case-2: Field in Nested Document equals a value:**

-----

**Q1. Select all documents from the books collection where author profile contains 2 courses?**

**> db.books.find({"author.profile.courses": {\$eq: 2}}).pretty()**

**> db.books.find({"author.profile.courses": 2}).pretty()**

**Case-3: Array element equals a value:**

-----

**Q1. Read all documents from the books collection where 'tags' array contains 'database' element?**

**> db.books.find({tags: {\$eq: "database"}}).pretty()**

**> db.books.find({tags: "database"}).pretty()**

**Case-4: Equals Array Value directly:**

-----

**Q1. Select all documents from books collection where tags array is exactly equal to ["language","freeware","programming"].**

```
> db.books.find({tags: {$eq:
["language","freeware","programming"]}}).pretty()
> db.books.find({tags: ["language","freeware","programming"]}).pretty()
```

**\$ne operator:**

-----

**ne ---> means not equal**

**We can use \$ne operator to select all the documents where the value of the field is not equal to specified value.**

**Syntax: db.collection.find({field: {\$ne: value}})**

**Q. To select all documents from books collection where no\_of\_reviews is not equal to 3.**

```
> db.books.find({no_of_reviews: {$ne: 3}}).pretty()
```

**Note: If the specified field not available, such documents also will be included in the result.**

**\$gt operator:**

-----

**\$gt ---> means greater than**

**We can use \$gt operator to select all documents where the value of field is greater than specified value.**

**Syntax: db.collection.find({field: {\$gt: value}})**

**Q1. Select all documents from books collection where the no\_of\_reviews is greater than 3.**

**> db.books.find({no\_of\_reviews: {\$gt: 3}}).pretty()**

**\$gte operator:**

-----

**\$gte ----> means greater than or equal to**

**We can use \$gte operator to select all documents where the value of field is greater than or equal to specified value.**

**Syntax: db.collection.find({field: {\$gte: value}})**

**Q1. Select all documents from books collection where the no\_of\_reviews is greater than or equal to 3.**

**> db.books.find({no\_of\_reviews: {\$gte: 3}}).pretty()**

**\$lt operator:**

-----

**\$lt ---> means less than**

**We can use \$lt operator to select all documents where the value of field is less than specified value.**

**Syntax: db.collection.find({field: {\$lt: value}})**

**Q1. Select all documents from books collection where the no\_of\_reviews is less than 3.**

```
> db.books.find({no_of_reviews: {$lt: 3}}).pretty()
```

**\$lte operator:**

-----

**\$lte --> means less than or equal to**

**We can use \$lte operator to select all documents where the value of field is less than or equal to specified value.**

**Syntax: db.collection.find({field: {\$lte: value}})**

**Q1. Select all documents from books collection where the no\_of\_reviews is less than or equal to 3.**

```
> db.books.find({no_of_reviews: {$lte: 3}}).pretty()
```

**\$in operator:**

-----

**We can use \$in operator to select all documents where the value of a field equals any value in the specified array.**

**It is something like python membership operator.**

**Syntax: db.collection.find({field: {\$in: [value1,value2,...,valueN]}})**

**Q1. Select all documents from the books collection where the no\_of\_reviews is 1 or 4 or 5?**

```
> db.books.find({no_of_reviews: {$in: [1,4,5]}}).pretty()
```

**Q2. Select all documents from the books collection where the tags array contains either database or programming.**

```
> db.books.find({tags: {$in: ["database", "programming"]}}).pretty()
```

**\$nin operator:**

-----

**\$nin means not in**

**It is inverse of in operator**

**Syntax: db.collection.find({field: {\$nin: [value1,value2,...,valueN]}})**

**We can use \$nin operator to select all documents where:**

- 1. The field value not present in the specified array.**
- 2. The field does not exist.**

**Q1. Select all documents from books collection where the no\_of\_reviews is not 1 or 4 or 5?**

```
> db.books.find({no_of_reviews: {$nin: [1, 4, 5]}}).pretty()
```

**Q2. consider the query**

```
> db.books.find({exams: {$nin: [1, 4, 5]}}).pretty()
```

**We will get all documents, because exams field is not available in any document.**

**Note:**

**\$in result + \$nin result = total no of documents**

```
> db.books.find({no_of_reviews: {$nin: [1, 4, 5]}}).count()
```

**4**



```
> db.books.find({no_of_reviews: {$in: [1, 4, 5]}}).count()
4
> db.books.find().count()
8
```

### **Logical Query Operators:**

-----

**\$or, \$nor, \$and, \$not**

**oldtamirindrecepie**

**olden days tammarend pickle**

**sir take session on procrastination or time management**

**If you are not having goal in your life, you will part of some other's goal.**

**One Movie Dialogue sir ,If we not working for our goals then others will use to full fill their Dreamzs.**

### **Logical Query Operators:**

-----

**\$or, \$nor, \$and, \$not**

**\$or operator:**

-----

**\$or performs logical OR operation on an array of two or more expressions(conditions) and selects the documents that satisfy atleast one of the expression(condition)**

**Syntax: {\$or: [{expression1},{expression1},..{expressionN}]}**

**Q1. Select all documents where either no\_of\_reviews >3 or tags array contains programming element?**

**c1: {no\_of\_reviews: {\$gt: 3}}**

**c2: {tags: "programming"}**

**> db.books.find({\$or: [{no\_of\_reviews: {\$gt: 3}}, {tags: "programming"}]}).pretty()**

**Q2. Select all documents where either no\_of\_reviews is less than 3 or downloadable is true or author profile contains atleast 2 books?**

**c1: {no\_of\_reviews: {\$lt: 3}}**

**c2: {downloadable: true}**

**c3: {"author.profile.books": {\$gte: 2}}**

**> db.books.find({\$or: [{no\_of\_reviews: {\$lt: 3}}, {downloadable: true}, {"author.profile.books": {\$gte: 2}}]}).pretty()**

**\$nor operator:**

**-----**

**It is inverse of \$or operator.**

**\$or --->Atleast one condition satisfied**

**\$nor --->neither condition satisfied i.e all conditions fails**

**Syntax: {\$nor: [{expression1},{expression1},...{expressionN}]}**

**\$nor performs a logical NOR operation on an array of one or more expressions(conditions) and selects the documents that fail all query expressions in the array.**

**eg:**

**c1: { no\_of\_reviews: {\$gt: 3}}**

**c2: { downloadable: true}**

```
> db.books.find({$nor: [{ no_of_reviews: {$gt: 3}}, { downloadable: true}]}).pretty()
```

**It will select all documents where**

**1. The no\_of\_reviews is less than or equal to 3 (i.e not greater than 3)**

**AND**

**2. downloadable is false**

**3. documents does not contain no\_of\_reviews and downloadable fields**

**Note: \$or results + \$nor results = total no of documents**

```
> db.books.find({$or: [{ no_of_reviews: {$gt: 3}}, { downloadable: true}]}).count()
```

**5**

```
> db.books.find({$nor: [{ no_of_reviews: {$gt: 3}}, { downloadable: true}]}).count()
```

**3**

```
> db.books.find().count()
```

**8**

-----  
**\$and operator:**  
-----

**\$and performs logical AND operation on an array of one or more expressions and selects the documents that satisfy all expressions in the array. i.e all conditions should be satisfied.**

**Syntax: {\$and: [{expression1},{expression1},...{expressionN}]}**

**Q. Select all documents where the no\_of\_reviews is greater than or equals to 3 and downloadable is true?**

**c1: {no\_of\_reviews: {\$gte: 3}}**

**c2: {downloadable: true}**

**> db.books.find({\$and: [{no\_of\_reviews: {\$gte: 3}}, {downloadable: true}]}).pretty()**

**Assignment:**  
**students.json:**  
-----

```
[
  {
    "name": "A",
    "marks": 10
  },
  {
    "name": "B",
    "marks": 20
  },
  {
    "name": "C",
    "marks": 30
  }
]
```

```
},  
{  
  "name": "D",  
  "marks": 40  
},  
{  
  "name": "E",  
  "marks": 50  
},  
{  
  "name": "F",  
  "marks": 60  
},  
{  
  "name": "G",  
  "marks": 70  
},  
{  
  "name": "H",  
  "marks": 80  
},  
{  
  "name": "I",  
  "marks": 90  
},  
{  
  "name": "J",  
  "marks": 100  
}  
]  
mongoimport --db storedb --collection students --file students.json --  
jsonArray
```

**Q1. Select all students where marks are less than 85 and greater than 45?**

**c1: {marks: {\$lt: 85}}**

**c2: {marks: {\$gt: 45}}**

**> db.students.find({\$and: [{marks: {\$lt: 85}}, {marks: {\$gt: 45}}]}).pretty()**

**Q2. Select all students where marks are less than 50 and greater than or equal to 35?**

**c1: {marks: {\$lt: 50}}**

**c2: {marks: {\$gte: 35}}**

**> db.students.find({\$and: [{marks: {\$lt: 50}}, {marks: {\$gte: 35}}]}).pretty()**

**Shortcut for AND operation:**

-----

**MongoDB provides an implicit AND operation when specifying a comma separated list of expressions.**

**Normal way: {\$and: [{expression1},{expression1},...]}**

**Shortcut way: {expression1, expression2,... }**

**Q. Select all documents where the no\_of\_reviews is greater than or equals to 3 and downloadable is true?**

**c1: {no\_of\_reviews: {\$gte: 3}}**

**c2: {downloadable: true}**

**> db.books.find({\$and: [{no\_of\_reviews: {\$gte: 3}}, {downloadable: true}]}).pretty()**

**shortcut way:**

**> db.books.find({no\_of\_reviews: {\$gte: 3}, downloadable: true}).pretty()**

### **Limitation of this shortcut:**

-----

**If the conditions are on the same field then this short cut won't work.**

**Q. List out all students whose marks are  $\geq 50$  and  $\leq 90$ ?**

```
> db.students.find({marks: {$gte: 50}, marks: {$lte: 90}}).pretty()
{ "_id" : ObjectId("5fed4de50df1d8f23c0f678c"), "name" : "B", "marks" :
20 }
{ "_id" : ObjectId("5fed4de50df1d8f23c0f678d"), "name" : "I", "marks" :
90 }
{ "_id" : ObjectId("5fed4de50df1d8f23c0f678e"), "name" : "C", "marks" :
30 }
{ "_id" : ObjectId("5fed4de50df1d8f23c0f678f"), "name" : "A", "marks" :
10 }
{ "_id" : ObjectId("5fed4de50df1d8f23c0f6791"), "name" : "E", "marks" :
50 }
{ "_id" : ObjectId("5fed4de50df1d8f23c0f6792"), "name" : "G", "marks" :
70 }
{ "_id" : ObjectId("5fed4de50df1d8f23c0f6793"), "name" : "D", "marks" :
40 }
{ "_id" : ObjectId("5fed4de50df1d8f23c0f6794"), "name" : "F", "marks" :
60 }
{ "_id" : ObjectId("5fed4de50df1d8f23c0f6795"), "name" : "H", "marks" :
80 }
```

**Reason: In Javascript object, duplicate keys are not allowed. If we are trying to add duplicate keys then old value will be replaced with new value.**

```
{marks: {$gte: 50}, marks: {$lte: 90}}
```

**It will become**

**{marks: {\$lte: 90}}**

**Solution: we should use \$and operator**

```
> db.students.find({ $and: [ {marks: {$gte: 50}}, {marks: {$lte: 90}} ] }).pretty()
{ "_id" : ObjectId("5fed4de50df1d8f23c0f678d"), "name" : "I", "marks" : 90 }
{ "_id" : ObjectId("5fed4de50df1d8f23c0f6791"), "name" : "E", "marks" : 50 }
{ "_id" : ObjectId("5fed4de50df1d8f23c0f6792"), "name" : "G", "marks" : 70 }
{ "_id" : ObjectId("5fed4de50df1d8f23c0f6794"), "name" : "F", "marks" : 60 }
{ "_id" : ObjectId("5fed4de50df1d8f23c0f6795"), "name" : "H", "marks" : 80 }
```

-----

**Q. What is the difference between these two queries?**

- 1. db.students.find({ \$and: [ {marks: {\$gte: 50}}, {marks: {\$lte: 90}} ] }).pretty()**
- 2. db.students.find({marks: {\$gte: 50, \$lte: 90}}).pretty()**

**Both will provide same results.**

**In this we are using field only once and shortcut also will work.**

**\$not operator:**

-----

**It is just to perform inverse operation.**

**Syntax: { field: {\$not: {operator expression}}}**

**eg: { marks: {\$not: {\$gte: 10}}}**



**\$not operation performs logical NOT operation on the specified operator expression and selects the documents that do not match operator expression. This includes the documents that do not contain the field.**

**eg:**

**c1: { no\_of\_reviews: {\$gt: 3}}**

**db.books.find({ no\_of\_reviews: {\$not: {\$gt: 3}}}).pretty()**

**It returns all documents where**

- 1. The no\_of\_reviews not greater than 3( i.e less than or equal to 3)**
- 2. no\_of\_reviews field does not exist.**

### **Element Query Operators:**

-----

- 1. \$exists**
- 2. \$type**

#### **1. \$exists:**

-----

**Syntax: {field: {\$exists: <boolean>}}**

**If <boolean> is true, then it selects all documents that contain specified field even the value of the field is null.**

**If <boolean> is false, then it selects all documents that do not contain specified field.**

**Q1. Select all documents which contains no\_of\_reviews field.**

**> db.books.find({no\_of\_reviews: {\$exists: true}}).pretty()**

**Q2. Select all documents which does not contain no\_of\_reviews field.**

```
> db.books.find({no_of_reviews: {$exists: false}}).pretty()
```

**case study:**

-----

```
db.students.insertOne({name: "Durga", marks: 100, gf: "Sunny"})
```

```
db.students.insertOne({name: "Ravi", marks: 20, gf: "Mallika"})
```

**Q1. Select all students who are having gf field?**

```
> db.students.find({ gf: {$exists: true}}).pretty()
```

**Q2. Select all students who are not having gf field?**

```
> db.students.find({ gf: {$exists: false}}).pretty()
```

**Q3. Select all students who are having the gf, but still marks are greater than 70?**

```
c1: {gf: {$exists: true}}
```

```
c2: { marks: {$gt: 70}}
```

```
> db.students.find({ gf: {$exists: true}, marks: {$gt: 70} }).pretty()
```

```
> db.students.find({ $and: [{gf: {$exists: true}}, { marks: {$gt: 70}}  
] }).pretty()
```

+

-----

**2. \$type operator:**

-----

**\$type operator selects the documents where the value of the field is of a particular type.**

**We have to specify the type as BSON type.**

**This operator is very helpful, whenever we are dealing with large volumes of unstructured data where types are unpredictable.**

#### **Syntax-1: Querying for a single type**

-----

**{field: {\$type: <BSONType>}}**

**We can specify either number or alias for the BSON Type.  
eg:**

**{field: {\$type: "int"}}**

**{field: {\$type: "string"}}**

#### **Syntax-2: Querying for multiple types**

-----

**{field: {\$type: [<BSONType1>, <BSONType2>, <BSONType3>, ...]}}**

#### **Table:**

**BSON Type----->Number----->alias**

=====

**Double----->1----->"double"**

**String----->2----->"string"**

**Object----->3----->"object"**

**Array----->4----->"array"**

**BinaryData----->5----->"binData"**

**ObjectId----->7----->"objectId"**

**Boolean----->8----->"bool"**

**Date----->9----->"date"**

**Null----->10----->"null"**

**32 Bit Integer----->16----->"int"**

**64 Bit Integer----->18----->"long"**

**Decimal128----->19----->"decimal"**

**Q1. What is the difference between int and long?**

-----

**int --->32 bits integer value**

**long --->64 bits integer value**

**Q2. What is the difference between double and decimal?**

-----

**double --->64 bits floating point value**

**decimal --->128 bits floating point value**

**Note:**

**\$type supports "number" alias, which will match the following BSON Types.**

**int**

**long**

**double**

**decimal**

**Case Study:**

-----

**db.phonebook.insertOne({\_id: 1, name: "Sunny", phoneNumber: "9292929292"})**

**db.phonebook.insertOne({\_id: 2, name: "Bunny", phoneNumber: 8896979797})**

**db.phonebook.insertOne({\_id: 3, name: "Chinny", phoneNumber: NumberLong(9898989898) })**

**db.phonebook.insertOne({\_id: 4, name: "Vinny", phoneNumber: NumberInt(9246212143)})**

**db.phonebook.insertOne({\_id: 5, name: "Pinny", phoneNumber: ["8885252627", 8096969696]})**

**Every number is by default treated as double type in MongoDB.**

**"9292929292" --->string type**

**8896979797 ---->double type**

**NumberLong(9898989898) --->long type**

**NumberInt(9246212143) --->int type**

**Q1. Select all documents where phoneNumber value is of string type?**

```
> db.phonebook.find({phoneNumber: {$type: "string"}}).pretty()
```

```
> db.phonebook.find({phoneNumber: {$type: 2}}).pretty()
```

```
> db.phonebook.find({phoneNumber: {$type: 2}}).pretty()
```

```
{ "_id" : 1, "name" : "Sunny", "phoneNumber" : "9292929292" }  
{  
  "_id" : 5,  
  "name" : "Pinny",  
  "phoneNumber" : [  
    "8885252627",  
    8096969696  
  ]  
}
```

**Q2. Select all documents where phoneNumber value is of double type?**

```
> db.phonebook.find({phoneNumber: {$type: "double"}}).pretty()
```

```
> db.phonebook.find({phoneNumber: {$type: 1}}).pretty()
```

```
> db.phonebook.find({phoneNumber: {$type: 1}}).pretty()
```

```
{ "_id" : 2, "name" : "Bunny", "phoneNumber" : 8896979797 }  
{  
  "_id" : 5,
```

```

    "name" : "Pinny",
    "phoneNumber" : [
        "8885252627",
        8096969696
    ]
}

```

**Q3. Select all documents where phoneNumber value is of int type?**

```

> db.phonebook.find({phoneNumber: {$type: "int"}}).pretty()
> db.phonebook.find({phoneNumber: {$type: 16}}).pretty()

```

```

> db.phonebook.find({phoneNumber: {$type: 16}}).pretty()
{ "_id" : 4, "name" : "Vinny", "phoneNumber" : 656277551 }

```

**Note: NumberInt(9246212143) -->656277551**

**9246212143 cannot be accomodated in 32 bits. Hence some loss of information.**

**Q4. Select all documents where phoneNumber value is of long type?**

```

> db.phonebook.find({phoneNumber: {$type: "long"}}).pretty()
> db.phonebook.find({phoneNumber: {$type: 18}}).pretty()

```

```

> db.phonebook.find({phoneNumber: {$type: 18}}).pretty()
{ "_id" : 3, "name" : "Chinny", "phoneNumber" :
NumberLong("9898989898") }

```

**Q5. Select all documents where phoneNumber value is of number type?**

```
> db.phonebook.find({phoneNumber: {$type: "number"}}).pretty()
```

```
{ "_id" : 2, "name" : "Bunny", "phoneNumber" : 8896979797 }  
{ "_id" : 3, "name" : "Chinny", "phoneNumber" :  
NumberLong("9898989898") }  
{ "_id" : 4, "name" : "Vinny", "phoneNumber" : 656277551 }  
{  
  "_id" : 5,  
  "name" : "Pinny",  
  "phoneNumber" : [  
    "8885252627",  
    8096969696  
  ]  
}
```

**Q6. Querying by multiple data types**

**Select all documents where phoneNumber value is of either string or double.**

```
> db.phonebook.find({phoneNumber: {$type: ["string",  
"double"]}}).pretty()
```

```
> db.phonebook.find({phoneNumber: {$type: [2, 1]}}).pretty()
```

```
> db.phonebook.find({phoneNumber: {$type: [2, 1]}}).pretty()  
{ "_id" : 1, "name" : "Sunny", "phoneNumber" : "9292929292" }  
{ "_id" : 2, "name" : "Bunny", "phoneNumber" : 8896979797 }  
{  
  "_id" : 5,  
  "name" : "Pinny",  
  "phoneNumber" : [  
    "8885252627",
```

**8096969696**

**]  
}**

**comparison operators: \$gt,\$gte,\$lt,\$lte,\$eq,\$ne,\$in,\$nin**

**logical operators: \$or, \$nor, \$and, \$not**

**element query operators: \$exists, \$type**

**Evaluation Query Operators:**

-----

**The operators which can be used for evaluation purposes are called Evaluation Query Operators.**

**1. \$expr 2. \$regex 3. \$mod 4. \$jsonSchema 5. \$text 6. \$where**

**1. \$expr operator:**

-----

**expr means expression.**

**Evaluate expression and select documents which satisfy that expression.**

**Syntax:**

**{ \$expr: {<expression>}}**

**It is very helpful to compare two field values within the same document.**

**Case Study: Compare two field values from the same document:**

-----

**db.homeBudget.insertOne({category:"home food", budget:1000, spent:1500})**

**db.homeBudget.insertOne({category:"outside food", budget:1000, spent:2000})**



```
db.homeBudget.insertOne({category:"rent", budget:1500, spent:1500})
db.homeBudget.insertOne({category:"education", budget:2000,
spent:1000})
db.homeBudget.insertOne({category:"clothes", budget:750,
spent:1500})
db.homeBudget.insertOne({category:"entertainment", budget:1000,
spent:2500})
```

**Q1. Select all documents where spent amount exceeds budget amount?**

```
> db.homeBudget.find({$expr: {$gt: ['$spent','$budget']}}).pretty()
{
  "_id" : ObjectId("5ff52c2345f8edf724d263ec"),
  "category" : "home food",
  "budget" : 1000,
  "spent" : 1500
}
{
  "_id" : ObjectId("5ff52c2345f8edf724d263ed"),
  "category" : "outside food",
  "budget" : 1000,
  "spent" : 2000
}
{
  "_id" : ObjectId("5ff52c2345f8edf724d263f0"),
  "category" : "clothes",
  "budget" : 750,
  "spent" : 1500
}
{
  "_id" : ObjectId("5ff52c2645f8edf724d263f1"),
  "category" : "entertainment",
  "budget" : 1000,
```

```
    "spent" : 2500
}
```

**Q2. Select all documents where spent amount is less than or equal to budget amount?**

```
> db.homeBudget.find({$expr: {$lte: ["$spent","$budget"]}}).pretty()
{
  "_id" : ObjectId("5ff52c2345f8edf724d263ee"),
  "category" : "rent",
  "budget" : 1500,
  "spent" : 1500
}
{
  "_id" : ObjectId("5ff52c2345f8edf724d263ef"),
  "category" : "education",
  "budget" : 2000,
  "spent" : 1000
}
```

**Q3. Select all documents where spent amount is equal to budget amount?**

```
> db.homeBudget.find({$expr: {$eq: ["$spent","$budget"]}}).pretty()
{
  "_id" : ObjectId("5ff52c2345f8edf724d263ee"),
  "category" : "rent",
  "budget" : 1500,
  "spent" : 1500
}
```

**Note: \$expr operator is very commonly used with aggregation expressions.**

## **2. \$regex operator:**

-----

**regex means regular expression.**

**We can use \$regex operator to select documents where values match a specified regular expression.**

### **Syntax:**

-----

**We can use \$regex operator in any of the following styles:**

**{ field: { \$regex: /pattern/, \$options:'<options>'}}**

**{ field: { \$regex: 'pattern', \$options:'<options>'}}**

**{ field: { \$regex: /pattern/<options>'}}**

**{ field: /pattern/<options>'}}**

### **Case Study:**

-----

**db.homeBudget.insertOne({category:"home food", budget:1000, spent:1500})**

**db.homeBudget.insertOne({category:"outside food", budget:1000, spent:2000})**

**db.homeBudget.insertOne({category:"rent", budget:1500, spent:1500})**

**db.homeBudget.insertOne({category:"education", budget:2000, spent:1000})**

**db.homeBudget.insertOne({category:"clothes", budget:750, spent:1500})**

**db.homeBudget.insertOne({category:"entertainment", budget:1000, spent:2500})**

**Q1. Select all documents where category value contains food?**

```

> db.homeBudget.find({ category: { $regex: /food/}}).pretty()
> db.homeBudget.find({ category: { $regex: 'food'}}).pretty()
> db.homeBudget.find({ category: /food/}).pretty()
{
  "_id" : ObjectId("5ff52c2345f8edf724d263ec"),
  "category" : "home food",
  "budget" : 1000,
  "spent" : 1500
}
{
  "_id" : ObjectId("5ff52c2345f8edf724d263ed"),
  "category" : "outside food",
  "budget" : 1000,
  "spent" : 2000
}

```

**Note:** It is something like 'like operator': '%xxx' or 'xxx%' or '%xxx%' in relational databases.

**Note:** We can use ^ symbol in regular expressions to indicate starts with.

**Q2. Select all documents where category value starts with 'e'?**

```

> db.homeBudget.find({ category: { $regex: /^e/}}).pretty()
> db.homeBudget.find({ category: { $regex: '^e'}}).pretty()
> db.homeBudget.find({ category: /^e/}).pretty()
{
  "_id" : ObjectId("5ff52c2345f8edf724d263ef"),
  "category" : "education",
  "budget" : 2000,
  "spent" : 1000
}

```

```

{
  "_id" : ObjectId("5ff52c2645f8edf724d263f1"),
  "category" : "entertainment",
  "budget" : 1000,
  "spent" : 2500
}

```

**Note:**

**[abc] --->either a or b or c**

**[ec] --->either e or c**

**^[ec] ---> starts with either e or c**

**Q3. Select all documents where category value starts with either 'e' or 'c'?**

```
> db.homeBudget.find({ category: { $regex: /^[ec]/}}).pretty()
```

```
> db.homeBudget.find({ category: { $regex: '^[ec]'}}).pretty()
```

```
> db.homeBudget.find({ category: /^[ec]/}).pretty()
```

```

{
  "_id" : ObjectId("5ff52c2345f8edf724d263ef"),
  "category" : "education",
  "budget" : 2000,
  "spent" : 1000
}

```

```

{
  "_id" : ObjectId("5ff52c2345f8edf724d263f0"),
  "category" : "clothes",
  "budget" : 750,
  "spent" : 1500
}
{
  "_id" : ObjectId("5ff52c2645f8edf724d263f1"),
  "category" : "entertainment",

```

```
    "budget" : 1000,  
    "spent" : 2500  
}
```

**Note:**

**^t ---> starts with t**

**t\$ ---> ends with t**

**Q4. Select all documents where category value ends with 't'?**

```
> db.homeBudget.find({ category: { $regex: /t$/}}).pretty()
```

```
> db.homeBudget.find({ category: { $regex: 't$'}}).pretty()
```

```
> db.homeBudget.find({ category: /t$/}).pretty()
```

```
{  
  "_id" : ObjectId("5ff52c2345f8edf724d263ee"),  
  "category" : "rent",  
  "budget" : 1500,  
  "spent" : 1500  
}  
{  
  "_id" : ObjectId("5ff52c2645f8edf724d263f1"),  
  "category" : "entertainment",  
  "budget" : 1000,  
  "spent" : 2500  
}
```

**Q. Select all documents where category value ends with either 't' or 'n'?**

```
> db.homeBudget.find({ category: { $regex: /[tn]$/}}).pretty()
```

```
> db.homeBudget.find({ category: { $regex: '[tn]$'}}).pretty()
```

```
> db.homeBudget.find({ category: /[tn]$/}).pretty()
```

```
{
```

```

    "_id" : ObjectId("5ff52c2345f8edf724d263ee"),
    "category" : "rent",
    "budget" : 1500,
    "spent" : 1500
  }
  {
    "_id" : ObjectId("5ff52c2345f8edf724d263ef"),
    "category" : "education",
    "budget" : 2000,
    "spent" : 1000
  }
  {
    "_id" : ObjectId("5ff52c2645f8edf724d263f1"),
    "category" : "entertainment",
    "budget" : 1000,
    "spent" : 2500
  }
}

```

**How to check case insensitivity:**

-----

**By default case will be considered. If we want to ignore case, ie if we want case insensitivity then we should use 'i' option.**

**i means case insensitive.**

**Q. Select all documents where category value starts with either e or E?**

```

> db.homeBudget.find({ category: {$regex: /^E/, $options: 'i'}}).pretty()
> db.homeBudget.find({ category: {$regex: '^E', $options: 'i'}}).pretty()
> db.homeBudget.find({ category: {$regex: /^E/i}}).pretty()
> db.homeBudget.find({ category: /^E/i}).pretty()
{
  "_id" : ObjectId("5ff52c2345f8edf724d263ef"),

```

```

    "category" : "education",
    "budget" : 2000,
    "spent" : 1000
  }
  {
    "_id" : ObjectId("5ff52c2645f8edf724d263f1"),
    "category" : "entertainment",
    "budget" : 1000,
    "spent" : 2500
  }

```

### 3. \$mod operator:

-----

**mod means modulo operator or remainder operator.**

**It is very helpful to select documents based on modulo operation.**

**We can use \$mod operator to select documents where the value of the field divided by a divisor has a specified remainder.**

**Syntax: { field: {\$mod: [divisor, remainder]}}**

### Case Study:

-----

```

db.shop.insertOne({_id: 1, item: "soaps", quantity: 13})
db.shop.insertOne({_id: 2, item: "books", quantity: 10})
db.shop.insertOne({_id: 3, item: "pens", quantity: 15})
db.shop.insertOne({_id: 4, item: "pencils", quantity: 17})

```

**Q1. Select all documents of shop collection where quantity value is divisible by 5?**

```

> db.shop.find({ quantity: {$mod: [5, 0]}}).pretty()
{ "_id" : 2, "item" : "books", "quantity" : 10 }

```



```
{ "_id" : 3, "item" : "pens", "quantity" : 15 }
```

**Q2. Select all documents of shop collection where quantity value is divisible by 4 and has remainder 1.**

```
> db.shop.find({ quantity: {$mod: [4, 1]}}).pretty()  
{ "_id" : 1, "item" : "soaps", "quantity" : 13 }  
{ "_id" : 4, "item" : "pencils", "quantity" : 17 }
```

**Note: { field: {\$mod: [divisor, remainder]}}**

**Compulsory we have to provide both divisor and remainder, otherwise we will get error.**

**eg1:**

```
> db.shop.find({ quantity: {$mod: [4]}}).pretty()  
Error: error: {  
  "ok" : 0,  
  "errmsg" : "malformed mod, not enough elements",  
  "code" : 2,  
  "codeName" : "BadValue"  
}
```

**eg2:**

```
> db.shop.find({ quantity: {$mod: [4,1,2]}}).pretty()  
Error: error: {  
  "ok" : 0,  
  "errmsg" : "malformed mod, too many elements",  
  "code" : 2,  
  "codeName" : "BadValue"  
}
```

**4. \$jsonSchema:**

-----  
**We can use this operator to select documents based on given jsonSchema.**

#### **5. \$text:**

-----  
**It is related to indexes concept, will be discussed soon.**

#### **6. \$where:**

-----  
**It is deprecated and replaced \$expr.**

#### **Array Query Operators:**

-----  
**1. \$all 2. \$elemMatch 3. \$size**

#### **1. \$all operator:**

-----  
**We can use \$all operator to select documents where array contains all specified elements.**

#### **Syntax:**

-----  
**{ field: { \$all: [value1, value2, value3,...]}}**

**We can write equivalent query by using \$and operator also.**

**{ \$and: [{field: value1},{field: value2},{field: value3},...]}**

#### **Case Study:**

-----  
**db.courses.insertOne({\_id:1,  
name:"java",tags:["language","programming","easy","ocean"]})**

```

db.courses.insertOne({_id:2,
name:"python",tags:["language","programming","easy"]})
db.courses.insertOne({_id:3,
name:"C",tags:["language","performance"]})
db.courses.insertOne({_id:4,
name:"Oracle",tags:["database","sql","cloud"]})
db.courses.insertOne({_id:5,
name:"MongoDB",tags:["database","nosql","cloud"]})
db.courses.insertOne({_id:6, name:"Devops",tags:["culture"]})

```

**Q1. Select all documents where tags array contains "database" and "cloud" elements?**

```

> db.courses.find({$and: [{tags: "database"}, {tags: "cloud"}]}).pretty()
> db.courses.find({tags: {$all: ["database","cloud"]}}).pretty()
{
  "_id" : 4,
  "name" : "Oracle",
  "tags" : [
    "database",
    "sql",
    "cloud"
  ]
}
{
  "_id" : 5,
  "name" : "MongoDB",
  "tags" : [
    "database",
    "nosql",
    "cloud"
  ]
}

```

**Note: Order of elements is not important and it is not exact match.**

**> db.courses.find({tags: ["database","cloud"]}).pretty()===>Here order is important and Exact Match**

**Q2. Select all documents where tags array contains "language" and "programming" elements?**

```
> db.courses.find({$and: [{tags: "language"}, {tags: "programming"}]}).pretty()
> db.courses.find({tags: {$all: ["language","programming"]}}).pretty()
{
  "_id" : 1,
  "name" : "java",
  "tags" : [
    "language",
    "programming",
    "easy",
    "ocean"
  ]
}
{
  "_id" : 2,
  "name" : "python",
  "tags" : [
    "language",
    "programming",
    "easy"
  ]
}
```

**2. \$elemMatch Operator:**

-----  
**elemMatch means element Match.**

**We can use \$elemMatch operator to select documents where atleast one element of the array matches the specified query criteria.**

**Syntax:**

**{field: {\$elemMatch: {<query1>,<query2>,<query3>,...}}}**

**Case Study:**

-----  
**db.students.insertOne({\_id:1,name:"Durga",marks:[82,35,99]})  
db.students.insertOne({\_id:2,name:"Ravi",marks:[75,90,95]})**

**Q1. Select documents where student has atleast one subject marks greater than or equal to 80 but less than 90?**

**> db.students.find({marks: {\$elemMatch: {\$gte: 80, \$lt: 90}}}).pretty()  
{ "\_id" : 1, "name" : "Durga", "marks" : [ 82, 35, 99 ] }**

**82 is greater than or equal to 80 but less than 90.**

**3. \$size operator:**

-----  
**We can use \$size operator to select documents based on specified array size.**

**Syntax: { field: {\$size: n} }**

**Case Study:**

```
db.courses.insertOne({_id:1,
name:"java",tags:["language","programming","easy","ocean"]})
db.courses.insertOne({_id:2,
name:"python",tags:["language","programming","easy"]})
db.courses.insertOne({_id:3,
name:"C",tags:["language","performance"]})
db.courses.insertOne({_id:4,
name:"Oracle",tags:["database","sql","cloud"]})
db.courses.insertOne({_id:5,
name:"MongoDB",tags:["database","nosql","cloud"]})
db.courses.insertOne({_id:6, name:"Devops",tags:["culture"]})
```

**Q1. Select all documents where tags array contains exactly 4 elements?**

```
> db.courses.find({tags: {$size: 4}}).pretty()
{
  "_id" : 1,
  "name" : "java",
  "tags" : [
    "language",
    "programming",
    "easy",
    "ocean"
  ]
}
```

**Q2. Select all documents where tags array contains exactly 3 elements?**

```
> db.courses.find({tags: {$size: 3}}).pretty()
{
  "_id" : 2,
  "name" : "python",
  "tags" : [
```

```

        "language",
        "programming",
        "easy"
    ]
}
{
    "_id" : 4,
    "name" : "Oracle",
    "tags" : [
        "database",
        "sql",
        "cloud"
    ]
}
{
    "_id" : 5,
    "name" : "MongoDB",
    "tags" : [
        "database",
        "nosql",
        "cloud"
    ]
}

```

**Q3. Select all documents where tags array contains exactly 1 element?**

```

> db.courses.find({tags: {$size: 1}}).pretty()
{ "_id" : 6, "name" : "Devops", "tags" : [ "culture" ] }

```

**Note: \$size does not accept range of values.**

**How to import data from csv file to MongoDB?**

-----  
**vidya.csv:**

-----  

<b>empno</b>	<b>name</b>	<b>salary</b>
<b>1</b>	<b>Vidya</b>	<b>5000</b>
<b>2</b>	<b>Ravi</b>	<b>2000</b>
<b>3</b>	<b>Durga</b>	<b>6000</b>
<b>4</b>	<b>Sushma</b>	<b>3000</b>

**Command: mongoimport -d storedb -c emp --type csv --headerline --drop vidya.csv**

**> show collections**

**books**

**courses**

**emp**

**homeBudget**

**learners**

**phonebook**

**shop**

**students**

**> db.emp.find().pretty()**

```
{
  "_id" : ObjectId("5ff7cfe9ba62f324ceb5df09"),
  "empno" : 1,
  "name" : "Vidya",
  "salary" : 5000
}
{
  "_id" : ObjectId("5ff7cfe9ba62f324ceb5df0a"),
  "empno" : 2,
  "name" : "Ravi",
  "salary" : 2000
}
```



```

}
{
  "_id" : ObjectId("5ff7cfe9ba62f324ceb5df0b"),
  "empno" : 4,
  "name" : "Sushma",
  "salary" : 3000
}
{
  "_id" : ObjectId("5ff7cfe9ba62f324ceb5df0c"),
  "empno" : 3,
  "name" : "Durga",
  "salary" : 6000
}

```

**eg-2:**

-----

**learners.csv:**

-----

<b>_id</b>	<b>name</b>	<b>marks</b>
<b>1</b>	<b>narayan pradhan</b>	<b>10</b>
<b>2</b>	<b>abhilash</b>	<b>20</b>
<b>3</b>	<b>rasika</b>	<b>30</b>
<b>4</b>	<b>pankaj bhandari</b>	<b>40</b>
<b>5</b>	<b>Sheshanand Singh</b>	<b>50</b>
<b>6</b>	<b>dhanaraju</b>	<b>60</b>
<b>7</b>	<b>Satyasundar Panigrahi</b>	<b>70</b>
<b>8</b>	<b>jyothi</b>	<b>80</b>

**Command: mongoimport -d storedb -c learners --type csv --headerline --drop learners.csv**

**Q. How to insert documents from csv file**

**Command 1:**

```
mongoimport --db csvdb --collection sample_csv --file  
"Sample_csv_file.csv" --type csv --headerline
```

**# --headerline is used to ensure that our csv headerline should be read as keys while importing into database**

**Command 2:**

```
mongoimport --db csvdb --collection sample_csv --file  
"Sample_csv_file.csv" --type csv --headerline --maintainInsertionOrder
```

**# --maintainInsertionOrder ensures that the data will be inserted in the top to bottom order from the csv file (row by row)**

**Command 3:**

```
mongoimport --db csvdb --collection sample_csv --file  
"Sample_csv_file.csv" --type csv --headerline --maintainInsertionOrder --  
ignoreBlanks
```

**# --ignoreBlanks ---> Whereever the value for key is blank inside csv file, that particular key itself will not be imported to database for that particular document**

**The Complete Story of Cursor concept:**

-----

**Diagram**

**In a collection there may be a chance of lakhs of documents. Whenever we are trying to retrieve data from database, if MongoDB server sends total data, there may be a chance of the following problems:**

- 1. Storage problems**
- 2. Network traffic problem**
- 3. Performance problems**
- etc**

**To prevent these problems, most of the databases including MongoDB, uses cursor concept.**

**In MongoDB, if we are using find() method we won't get documents and we will get cursor object.**

**The return type of find() method is cursor object.**

**By using cursor object we can get data either batch wise or document wise.**

**By default cursor object will provide documents in batch wise. The default batch size is 20. But we can customize this value. For this we have to use DBQuery.shellBatchSize property.**

```
> db.learners.find().pretty()
{ "_id" : 2, "name" : "abhilash", "marks" : 20 }
{ "_id" : 3, "name" : "rasika", "marks" : 30 }
{ "_id" : 1, "name" : "narayan pradhan", "marks" : 10 }
{ "_id" : 6, "name" : "dhanaraju", "marks" : 60 }
{ "_id" : 7, "name" : "Satyasundar Panigrahi", "marks" : 70 }
{ "_id" : 5, "name" : "Sheshanand Singh", "marks" : 50 }
{ "_id" : 8, "name" : "jyothi", "marks" : 80 }
{ "_id" : 10, "name" : "bindhiya", "marks" : 100 }
{ "_id" : 9, "name" : "Hari", "marks" : 90 }
{ "_id" : 4, "name" : "pankaj bhandari", "marks" : 40 }
{ "_id" : 11, "name" : "vikas kale", "marks" : 10 }
{ "_id" : 13, "name" : "shashank sanap", "marks" : 30 }
{ "_id" : 12, "name" : "Sunita Kumati Choudhuri", "marks" : 20 }
{ "_id" : 15, "name" : "TharunK", "marks" : 50 }
```

```
{ "_id" : 14, "name" : "Atul", "marks" : 40 }
{ "_id" : 18, "name" : "Dusmant Kumar Mohapatra", "marks" : 80 }
{ "_id" : 16, "name" : "aron", "marks" : 60 }
{ "_id" : 24, "name" : "G.shukeshreddy", "marks" : 40 }
{ "_id" : 25, "name" : "Dakshesh", "marks" : 50 }
{ "_id" : 26, "name" : "Paramesh", "marks" : 60 }
```

Type "it" for more

> it

```
{ "_id" : 27, "name" : "Mahmodul Hasan", "marks" : 70 }
{ "_id" : 28, "name" : "ASHA", "marks" : 80 }
{ "_id" : 17, "name" : "pooja", "marks" : 70 }
{ "_id" : 30, "name" : "Maheshbabu", "marks" : 100 }
{ "_id" : 21, "name" : "Suraj Prasim Patel", "marks" : 10 }
{ "_id" : 32, "name" : "kusuma", "marks" : 20 }
{ "_id" : 33, "name" : "Vaishnavi Lende", "marks" : 30 }
{ "_id" : 19, "name" : "Deepak", "marks" : 90 }
{ "_id" : 23, "name" : "Zaid", "marks" : 30 }
{ "_id" : 20, "name" : "Bhim Kumar", "marks" : 100 }
{ "_id" : 22, "name" : "Naveen", "marks" : 20 }
{ "_id" : 35, "name" : "Prakash", "marks" : 50 }
{ "_id" : 31, "name" : "rajat kumar maurya", "marks" : 10 }
{ "_id" : 34, "name" : "Vengadesan", "marks" : 40 }
{ "_id" : 29, "name" : "Bharti Kardile", "marks" : 90 }
{ "_id" : 37, "name" : "sanat", "marks" : 70 }
{ "_id" : 38, "name" : "Aneesh Fathima", "marks" : 80 }
{ "_id" : 36, "name" : "raj", "marks" : 60 }
{ "_id" : 39, "name" : "Ratemo", "marks" : 90 }
{ "_id" : 41, "name" : "jignesh", "marks" : 10 }
```

Type "it" for more

```
> DBQuery.shellBatchSize = 5;
```

```
5
```

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

```
{ "_id" : 2, "name" : "abhilash", "marks" : 20 }  
{ "_id" : 3, "name" : "rasika", "marks" : 30 }  
{ "_id" : 1, "name" : "narayan pradhan", "marks" : 10 }  
{ "_id" : 6, "name" : "dhanaraju", "marks" : 60 }  
{ "_id" : 7, "name" : "Satyasundar Panigrahi", "marks" : 70 }
```

Type "it" for more

> it

```
{ "_id" : 5, "name" : "Sheshanand Singh", "marks" : 50 }  
{ "_id" : 8, "name" : "jyothi", "marks" : 80 }  
{ "_id" : 10, "name" : "bindhiya", "marks" : 100 }  
{ "_id" : 9, "name" : "Hari", "marks" : 90 }  
{ "_id" : 4, "name" : "pankaj bhandari", "marks" : 40 }
```

Type "it" for more

#### **Advantages of cursor:**

-----

- 1. We can get only required number of documents.**
- 2. We can get either batch wise or document wise.**
- 3. No chance of storage problems.**
- 4. No chance of network traffic problems.**
- 5. No chance of performance issues.**

#### **Important Methods of cursor:**

-----

##### **1. count()**

**To get total number of documents**

##### **2. hasNext()**

**To check whether the next document is available or not. If it is available then it returns true, otherwise returns false.**

##### **3. next()**

**To get next document. If there is no next document then we will get error.**

**Q. Why we are getting same document in the following case?**

```
> db.learners.find().next()
{ "_id" : 2, "name" : "abhilash", "marks" : 20 }
> db.learners.find().next()
{ "_id" : 2, "name" : "abhilash", "marks" : 20 }
```

**Ans: Here two different cursor objects we are using.**

```
var mycursor = db.learners.find()
mycursor.next()
mycursor.next()
```

**In this case we are using same cursor object and hence different documents we will get.**

```
> var mycursor = db.learners.find()
> mycursor.next()
{ "_id" : 2, "name" : "abhilash", "marks" : 20 }
> mycursor.next()
{ "_id" : 3, "name" : "rasika", "marks" : 30 }
> mycursor.next()
{ "_id" : 1, "name" : "narayan pradhan", "marks" : 10 }
```

**Javascript based code to get documents one by one:**

-----

**eg-1:**

```
> var mycursor = db.emp.find()
> mycursor.hasNext()
true
> mycursor.next()
{
  "_id" : ObjectId("5ff7cfe9ba62f324ceb5df09"),
  "empno" : 1,
  "name" : "Vidya",
  "salary" : 5000
}
> mycursor.next()
{
  "_id" : ObjectId("5ff7cfe9ba62f324ceb5df0a"),
  "empno" : 2,
  "name" : "Ravi",
  "salary" : 2000
}
```

**eg-2:**

```
var mycursor = db.learners.find();
while( mycursor.hasNext() )
{
  print(tojson(mycursor.next()));
}
```

**on shell:**

```
var mycursor = db.learners.find();
while( mycursor.hasNext() ) { print(tojson(mycursor.next())); }
```

**Note:** mycursor.next() returns **BSON** object. We have to convert **BSON** Object to json by using tojson() method.

**eg-3:**

-----

```
var mycursor = db.learners.find();  
while( mycursor.hasNext() )  
{  
    printjson(mycursor.next());  
}
```

**on shell:**

```
var mycursor = db.learners.find();  
while( mycursor.hasNext() ) { printjson(mycursor.next()); }
```

**eg-4:**

-----

```
var mycursor = db.learners.find();  
mycursor.forEach( doc => { printjson(doc) } )
```

**eg-5:**

-----

```
var mycursor = db.learners.find();  
mycursor.forEach(printjson)
```

**Cursor Helper Methods:**

-----

**We can use the following helper methods to shape our results:**

- 1. limit()**
- 2. skip()**
- 3. sort()**

**1. limit():**



-----  
**We can use this limit() method to limit the number of documents in the result.**

```
> db.learners.find().count()  
6571
```

```
> db.learners.find().limit(1)  
{ "_id" : 2, "name" : "abhilash", "marks" : 20 }
```

```
> db.learners.find().limit(2)  
{ "_id" : 2, "name" : "abhilash", "marks" : 20 }  
{ "_id" : 3, "name" : "rasika", "marks" : 30 }
```

```
> db.learners.find().limit(5)  
{ "_id" : 2, "name" : "abhilash", "marks" : 20 }  
{ "_id" : 3, "name" : "rasika", "marks" : 30 }  
{ "_id" : 1, "name" : "narayan pradhan", "marks" : 10 }  
{ "_id" : 6, "name" : "dhanaraju", "marks" : 60 }  
{ "_id" : 7, "name" : "Satyasundar Panigrahi", "marks" : 70 }
```

```
> db.learners.find().limit(25)  
{ "_id" : 2, "name" : "abhilash", "marks" : 20 }  
{ "_id" : 3, "name" : "rasika", "marks" : 30 }  
{ "_id" : 1, "name" : "narayan pradhan", "marks" : 10 }  
{ "_id" : 6, "name" : "dhanaraju", "marks" : 60 }  
{ "_id" : 7, "name" : "Satyasundar Panigrahi", "marks" : 70 }  
{ "_id" : 5, "name" : "Sheshanand Singh", "marks" : 50 }  
{ "_id" : 8, "name" : "jyothi", "marks" : 80 }  
{ "_id" : 10, "name" : "bindhiya", "marks" : 100 }  
{ "_id" : 9, "name" : "Hari", "marks" : 90 }  
{ "_id" : 4, "name" : "pankaj bhandari", "marks" : 40 }
```

```
{ "_id" : 11, "name" : "vikas kale", "marks" : 10 }
{ "_id" : 13, "name" : "shashank sanap", "marks" : 30 }
{ "_id" : 12, "name" : "Sunita Kumati Choudhuri", "marks" : 20 }
{ "_id" : 15, "name" : "TharunK", "marks" : 50 }
{ "_id" : 14, "name" : "Atul", "marks" : 40 }
{ "_id" : 18, "name" : "Dusmant Kumar Mohapatra", "marks" : 80 }
{ "_id" : 16, "name" : "aron", "marks" : 60 }
{ "_id" : 24, "name" : "G.shukeshreddy", "marks" : 40 }
{ "_id" : 25, "name" : "Dakshesh", "marks" : 50 }
{ "_id" : 26, "name" : "Paramesh", "marks" : 60 }
```

Type "it" for more

> it

```
{ "_id" : 27, "name" : "Mahmodul Hasan", "marks" : 70 }
{ "_id" : 28, "name" : "ASHA", "marks" : 80 }
{ "_id" : 17, "name" : "pooja", "marks" : 70 }
{ "_id" : 30, "name" : "Maheshbabu", "marks" : 100 }
{ "_id" : 21, "name" : "Suraj Prasim Patel", "marks" : 10 }
```

## 2. skip():

-----

We can use skip() method to skip the number of documents in the result.

> db.learners.find().skip(10)

To skip the first 10 documents.

**Q. To skip first 10 documents and to display next 10 documents?**

> db.learners.find().skip(10).limit(10)

## Use Case:

-----

In general we can use skip() and limit() methods in pagination concept while displaying our data.

**Assume per page 10 documents:**

**To display 1st page: `db.learners.find().limit(10)`**

**To display 2nd page: `db.learners.find().skip(10).limit(10)`**

**To display 3rd page: `db.learners.find().skip(20).limit(10)`**

**etc**

### **3. `sort()`:**

-----

**We can use `sort()` method to sort documents based on value of a particular field.**

**Syntax:**

**`sort({ field: 1})`**

**1 ==> means Ascending order/Alphabetical order**

**-1 ==> means Descending order/ Reverse of Alphabetical order**

**Q1. To display all learners based on ascending order of marks?**

**> `db.learners.find().sort({ marks: 1}).pretty()`**

**Q2. To display all learners based on descending order of marks?**

**> `db.learners.find().sort({ marks: -1}).pretty()`**

**Q3. To display all learners based on alphabetical order of names?**

**> `db.learners.find().sort({ name: 1}).pretty()`**

**Sorting based on multiple fields:**

-----

**We can sort based on multiple fields also.**

**Syntax: `sort({field1: 1, field2: 1,...})`**

**Sorting is based on field1, if field1 values are same then sorting based on field2 for those documents.**

**Q. Sort based on ascending order or marks. If two learners have same marks then sort based on reverse of alphabetical order of names?**

```
> db.learners.find().sort({ marks: 1, name: -1}).pretty()
```

**Note:**

-----

**Chaining of these helper methods is possible.**

```
> db.learners.find().sort({ name: -1}).skip(100).limit(15)
```

**All these methods will be executed from left to right and hence order is important.**

**Pagination based on alphabetical order of names:**

-----

**We have to sort based on alphabetical order of names. If two students having same name then consider ascending order of marks.**

**Per page only 15 documents.**

**1st page: db.learners.find().sort({name: 1, marks: 1}).limit(15).pretty()**

**2nd page: db.learners.find().sort({name: 1, marks: 1}).skip(15).limit(15).pretty()**

**3rd page: db.learners.find().sort({name: 1, marks: 1}).skip(30).limit(15).pretty()**

**etc**

**How to get documents with only required fields:**

-----

**We can get documents with only required fields instead of all fields.  
This is called projection.**

**Relational databases/sql databases:**  
-----

**without projection: select \* from employees;**

**with projection: select ename,esal from employees;**

**Projection in MongoDB?**  
-----

**db.collection.find({filter}) ==>without projection**

**db.collection.find({filter},{projection fields}) ==>with projection**

**Note: If we are providing projection list, compulsory we should provide filter object also, atleast empty java script object. i.e without providing first argument, we cannot talk about second argument.**

**eg: db.collection.find({}, {projection fields})**

**Case Study:**  
-----

**books collection: sample document**

```
{  
  "title": "Linux in simple way",  
  "isbn": 6677,  
  "downloadable": false,  
  "no_of_reviews": 1,  
  "tags": ["os","freeware","shell programming"],  
  "languages": ["english","hindi","telugu"],  
  "author": {  
    "name": "Shiva Ramachandran",
```

```

    "callname": "Shiv",
    "profile": {
        "exp":8,
        "courses":3,
        "books":2
    }
}

```

**db.collection.find({}, {projection fields})**

**Q1. To project only title and no\_of\_reviews?**

**> db.books.find({}, {title: 1, no\_of\_reviews: 1}).pretty()**

**field: 1 ==> means project/include this field in the result**

**field: 0 ==> means not to project/exclude this field in the result**

**If we are not taking any field in the projected list, by default that field will be excluded. ie default value is 0.**

**\_id field will be included always. But we can exclude this field by assigning with 0 explicitly.**

**> db.books.find({}, {title: 1, no\_of\_reviews: 1}).pretty()**

```

{
  "_id" : ObjectId("5fe95428fe935cdac43627c9"),
  "title" : "Java in simple way",
  "no_of_reviews" : 2
}
{
  "_id" : ObjectId("5fe95428fe935cdac43627ca"),
  "title" : "Linux in simple way",

```

```

    "no_of_reviews" : 1
  }
  {
    "_id" : ObjectId("5fe95428fe935cdac43627cb"),
    "title" : "MongoDB in simple way",
    "no_of_reviews" : 4
  }
  {
    "_id" : ObjectId("5fe95428fe935cdac43627cc"),
    "title" : "Python in simple way",
    "no_of_reviews" : 5
  }
  {
    "_id" : ObjectId("5fe95428fe935cdac43627cd"),
    "title" : "Shell Scripting in simple way",
    "no_of_reviews" : 1
  }
  {
    "_id" : ObjectId("5fe95428fe935cdac43627ce"),
    "title" : "Devops in simple way",
    "no_of_reviews" : 3
  }
  {
    "_id" : ObjectId("5fe95428fe935cdac43627cf"),
    "title" : "Oracle in simple way",
    "no_of_reviews" : 3
  }
}

```

**Note:**

```
> db.books.find({},{}).pretty()
```

**We will get all documents with all fields. Simply it is equals to:**

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

**Q2. To project only title and no\_of\_reviews without \_id ?**

```
> db.books.find({}, {title: 1, no_of_reviews: 1, _id: 0}).pretty()
```

```
{ "title" : "Java in simple way", "no_of_reviews" : 2 }  
{ "title" : "Linux in simple way", "no_of_reviews" : 1 }  
{ "title" : "MongoDB in simple way", "no_of_reviews" : 4 }  
{ "title" : "Python in simple way", "no_of_reviews" : 5 }  
{ "title" : "Shell Scripting in simple way", "no_of_reviews" : 1 }  
{ "title" : "Devops in simple way", "no_of_reviews" : 3 }  
{ "title" : "Oracle in simple way", "no_of_reviews" : 3 }
```

**Q3. Select all documents where no\_of\_reviews is greater than or equal to 3. Project only the following fields in every document?**

- 1. title**
- 2. no\_of\_reviews**
- 3. isbn**

```
> db.books.find({ no_of_reviews: { $gte: 3 } }, {title: 1, no_of_reviews: 1, isbn: 1, _id: 0 }).pretty()
```

```
{ "title" : "MongoDB in simple way", "isbn" : 6677, "no_of_reviews" : 4 }  
{ "title" : "Python in simple way", "isbn" : 1234, "no_of_reviews" : 5 }  
{ "title" : "Devops in simple way", "isbn" : 6677, "no_of_reviews" : 3 }  
{ "title" : "Oracle in simple way", "isbn" : 6677, "no_of_reviews" : 3 }
```

**Projection of Nested Document Fields:**

-----

**Q4. Project title, author's name and number of books in every document?**



```
> db.books.find({}, {title: 1, "author.name": 1, "author.profile.books": 1,
_id: 0 }).pretty()
```

```
{
  "title" : "Java in simple way",
  "author" : {
    "name" : "Karhik Ramachandran",
    "profile" : {
      "books" : 3
    }
  }
}
```

**Projection of arrays:**

-----

**Q. Project title, tags in every document of books collection?**

```
> db.books.find({}, { title: 1, tags: 1, _id: 0 }).pretty()
```

```
{
  "title" : "Java in simple way",
  "tags" : [
    "language",
    "freeware",
    "programming"
  ]
}
{
  "title" : "Linux in simple way",
  "tags" : [
    "os",
    "freeware",
    "shell programming"
  ]
}
```

}

## **Projection of Array Elements | Array Elements Projection Operators:**

---

```
> db.books.find({tags:"programming"}).pretty()
> db.books.find({tags:"programming",{title:1, tags:1, _id:0}).pretty()
> db.books.find({tags:"programming",{title:1, "tags.$":1, _id:0}).pretty()
```

**We can project array elements by using the following operators:**

- 1. \$**
- 2. \$elemMatch**
- 3. \$slice**

### **1. \$ Operator:**

---

**We can use \$ operator to project first element in an array that matches query condition.**

**Syntax:**

```
db.collection.find(<{<array>:<condition>,...}>,{<"<array>.$":1>})
```

### **Case Study:**

---

```
db.students.insertOne({_id:1, name:"Durga", year:1, marks:[70,87,90]})
db.students.insertOne({_id:2, name:"Ravi", year:1, marks:[90,88,92]})
db.students.insertOne({_id:3, name:"Shiva", year:1, marks:[85,100,90]})
db.students.insertOne({_id:4, name:"Durga", year:2, marks:[79,85,80]})
db.students.insertOne({_id:5, name:"Ravi", year:2, marks:[88,88,92]})
db.students.insertOne({_id:6, name:"Shiva", year:2, marks:[95,90,96]})
```

**Q1. db.students.find({marks:{\$gte: 85}},{\_id:0,marks:1})**

```
> db.students.find({marks:{$gte: 85}},{_id:0,marks:1})
{ "marks" : [ 70, 87, 90 ] }
{ "marks" : [ 90, 88, 92 ] }
{ "marks" : [ 85, 100, 90 ] }
{ "marks" : [ 79, 85, 80 ] }
{ "marks" : [ 88, 88, 92 ] }
{ "marks" : [ 95, 90, 96 ] }
```

**In this case all elements of array projected.**

**Q2. db.students.find({marks:{\$gte: 85}},{\_id:0,name: 1, "marks.\$":1})**

**Now instead of all elements, only first matched element will be projected.**

```
> db.students.find({marks:{$gte: 85}},{_id:0,name: 1, "marks.$":1})
{ "name" : "Durga", "marks" : [ 87 ] }
{ "name" : "Ravi", "marks" : [ 90 ] }
{ "name" : "Shiva", "marks" : [ 85 ] }
{ "name" : "Durga", "marks" : [ 85 ] }
{ "name" : "Ravi", "marks" : [ 88 ] }
{ "name" : "Shiva", "marks" : [ 95 ] }
```

**Q3. db.students.find({marks:{\$all: [88,90]}},{\_id:0,name: 1, "marks.\$":1})**

```
{ "name" : "Ravi", "marks" : [ 90 ] }
```

**Note: If there is no query condition or if query condition won't include array then we cannot use \$ operator, otherwise we will get error.**

**eg-1:**

```
> db.students.find({}, {_id:0,name: 1, "marks.$":1})
```

**Error: error: {**

```
"ok" : 0,  
"errmsg" : "positional operator '$' couldn't find a matching  
element in the array",  
"code" : 51246,  
"codeName" : "Location51246"  
}
```

**eg1:**

```
> db.students.find({year: 1},{_id:0,name: 1, "marks.$":1})
```

**Error: error: {**

```
"ok" : 0,  
"errmsg" : "positional operator '$' couldn't find a matching  
element in the array",  
"code" : 51246,  
"codeName" : "Location51246"  
}
```

**\*\*\*Note: \$ operator selects only one element which is first matched element based on query condition.**

## **2. \$elemMatch operator:**

**-----**

**1. selects only one element**

**2. which is matched element where condition is specified by \$elemMatch explicitly.**

**It never considers query condition.**

**We can use \$elemMatch to project first element in the array that matches specified \$elemMatch condition.**

**Q1.**

```
> db.students.find({}, {_id:0, name:1, year:1, marks:{$elemMatch:{$lt: 95}}})
```

```
{ "marks" : [ 70, 87, 90 ] }
```

```
{ "marks" : [ 90, 88, 92 ] }
```

```
{ "marks" : [ 85, 100, 90 ] }
```

```
{ "marks" : [ 79, 85, 80 ] }
```

```
{ "marks" : [ 88, 88, 92 ] }
```

```
{ "marks" : [ 95, 90, 96 ] }
```

```
{ "name" : "Durga", "year" : 1, "marks" : [ 70 ] }
```

```
{ "name" : "Ravi", "year" : 1, "marks" : [ 90 ] }
```

```
{ "name" : "Shiva", "year" : 1, "marks" : [ 85 ] }
```

```
{ "name" : "Durga", "year" : 2, "marks" : [ 79 ] }
```

```
{ "name" : "Ravi", "year" : 2, "marks" : [ 88 ] }
```

```
{ "name" : "Shiva", "year" : 2, "marks" : [ 90 ] }
```

```
> db.students.find({year:1}, {_id:0,
name:1, year:1, marks:{$elemMatch:{$gt: 85}}})
{ "name" : "Durga", "year" : 1, "marks" : [ 87 ] }
{ "name" : "Ravi", "year" : 1, "marks" : [ 90 ] }
{ "name" : "Shiva", "year" : 1, "marks" : [ 100 ] }
```

**What is the difference between \$ and \$elemMatch operators:**

-----

**Both operators project the first matching element from an array based on a condition.**

**\$ operator will select array element based on query condition. But \$elemMatch will select array element based on explicit condition specified by \$elemMatch but not based on query condition.**

```
> db.students.find({year:1,marks:{$gte: 85}},{_id:0,name:1,"marks.$":1})
{ "name" : "Durga", "marks" : [ 87 ] }
{ "name" : "Ravi", "marks" : [ 90 ] }
{ "name" : "Shiva", "marks" : [ 85 ] }
```

```
> db.students.find({year:1,marks:{$gte:
85}},{_id:0,name:1,marks:{$elemMatch:{$gt:89}}})
```

```
{ "name" : "Durga", "marks" : [ 90 ] }
{ "name" : "Ravi", "marks" : [ 90 ] }
{ "name" : "Shiva", "marks" : [ 100 ] }
```

### **3. \$slice operator:**

-----

**By using \$slice operator we can select required number of elements in the array.**

#### **Syntax-1:**

-----

```
db.collection.find({query},{<array>:{$slice: n}})
```

**n-->number of elements to be selected.**

**Specify a positive number n to return the first n elements.**

**Specify a negative number n to return the last n elements.**

**If n is greater than number of elements in the array then all elements will be selected.**

#### **eg-1:**

```
> db.students.find({},{_id:0,name:1,year:1, marks:{$slice: 2}})
```

**In the array only first 2 elements will be selected.**

```
{ "name" : "Durga", "year" : 1, "marks" : [ 70, 87 ] }  
{ "name" : "Ravi", "year" : 1, "marks" : [ 90, 88 ] }  
{ "name" : "Shiva", "year" : 1, "marks" : [ 85, 100 ] }  
{ "name" : "Durga", "year" : 2, "marks" : [ 79, 85 ] }  
{ "name" : "Ravi", "year" : 2, "marks" : [ 88, 88 ] }  
{ "name" : "Shiva", "year" : 2, "marks" : [ 95, 90 ] }
```

**eg-2:**

```
> db.students.find({}, {_id:0,name:1,year:1, marks:{$slice: -2}})
```

**In the array only last 2 elements will be selected.**

```
> db.students.find({}, {_id:0,name:1,year:1, marks:{$slice: -2}})
```

```
{ "name" : "Durga", "year" : 1, "marks" : [ 87, 90 ] }  
{ "name" : "Ravi", "year" : 1, "marks" : [ 88, 92 ] }  
{ "name" : "Shiva", "year" : 1, "marks" : [ 100, 90 ] }  
{ "name" : "Durga", "year" : 2, "marks" : [ 85, 80 ] }  
{ "name" : "Ravi", "year" : 2, "marks" : [ 88, 92 ] }  
{ "name" : "Shiva", "year" : 2, "marks" : [ 90, 96 ] }
```

**eg-3:**

```
> db.students.find({}, {_id:0,name:1,year:1, marks:{$slice: 100}})
```

**In this case all elements will be included.**

```
{ "name" : "Durga", "year" : 1, "marks" : [ 70, 87, 90 ] }  
{ "name" : "Ravi", "year" : 1, "marks" : [ 90, 88, 92 ] }  
{ "name" : "Shiva", "year" : 1, "marks" : [ 85, 100, 90 ] }  
{ "name" : "Durga", "year" : 2, "marks" : [ 79, 85, 80 ] }  
{ "name" : "Ravi", "year" : 2, "marks" : [ 88, 88, 92 ] }  
{ "name" : "Shiva", "year" : 2, "marks" : [ 95, 90, 96 ] }
```

**Syntax-2:**

-----

**db.collection.find({query},{<array>:{\$slice: [n1,n2]}})**

**skip n1 number of elements and then select n2 number of elements.**

**n1--->number to skip**

**n2--->number to return**

**eg-1:**

**skip first element and then select next two elements.**

**> db.students.find({year:1},{\_id:0,name:1, marks:{\$slice: [1,2]}})**

**{ "name" : "Durga", "marks" : [ 87, 90 ] }**

**{ "name" : "Ravi", "marks" : [ 88, 92 ] }**

**{ "name" : "Shiva", "marks" : [ 100, 90 ] }**

**eg-2: skip first 2 elements and select next 10 elements.**

**> db.students.find({year:1},{\_id:0,name:1, marks:{\$slice: [2,10]}})**

**{ "name" : "Durga", "marks" : [ 90 ] }**

**{ "name" : "Ravi", "marks" : [ 92 ] }**

**{ "name" : "Shiva", "marks" : [ 90 ] }**

**eg-3: required only 7th element in the array?**

**> db.students.find({}, {\_id:0,name:1, marks:{\$slice: [6,1]}})**

**eg-4: required from 3rd to 10th elements**

**> db.students.find({}, {\_id:0,name:1, marks:{\$slice: [2,8]}})**

## **CRUD Operations**

**C--->Create Operation | Insert Operation**

**R--->Retrieve Operation | Read Operation**

**U--->Update Operation**



## **U--->Update Operation:**

-----

**students collection**

**Based on our requirement, we can perform update operations to reflect latest information.**

**eg-1: update student document with changed mobile number**

**eg-2: Increment all employee salaries by 1000 if salary is less than 10000**

**We can perform updations like**

- 1. Overwrite existing value of a particular field with our new value**
  - 2. Add a new field for selected documents**
  - 3. Remove an existing field**
  - 4. Rename an existing field**
- etc**

## **How to perform updations:**

-----

**We can perform required updations by using update methods and update operators.**

## **update methods:**

-----

**There are 3 update methods are available.**

- 1. updateOne()**
- 2. updateMany()**
- 3. update()**

### **1. updateOne():**

-----

**db.collection.updateOne(filter,update,options)**

**It finds the first document that matches filter criteria and perform required updation. It will perform updation for a single document.**

### **2. updateMany():**

-----

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

**To update all documents that match the specified filter criteria.**

### **3. update():**

-----

**db.collection.update(filter,update,options)**

**We can use this method to update either a single document or multiple documents.**

**By default this method updates a single document only.**

**If we include multi:true to update all documents that match query criteria.**

**db.collection.update(filter,update)--->To update a single document**

**db.collection.update(filter,update,{multi:true})--->To update all matched documents**

### **Case Study:**

-----

**db.employees.insert({\_id:1,eno:100,ename:"Sunny",esal:1000,eaddr:"Mumbai"})**

```

db.employees.insert({_id:2,eno:200,ename:"Bunny",esal:2000,eaddr:"H
yderabad"})
db.employees.insert({_id:3,eno:300,ename:"Chinny",esal:3000,eaddr:"M
umbai"})
db.employees.insert({_id:4,eno:400,ename:"Vinny",esal:4000,eaddr:"De
lhi"})
db.employees.insert({_id:5,eno:500,ename:"Pinny",esal:5000,eaddr:"Ch
ennai"})
db.employees.insert({_id:6,eno:600,ename:"Tinny",esal:6000,eaddr:"Mu
mbai"})
db.employees.insert({_id:7,eno:700,ename:"Zinny",esal:7000,eaddr:"De
lhi"})

```

**Note: To perform updations we have to use update operators like \$set, \$unset, \$inc etc**

**\$set -->To set new value to the specified field**

**Q1. Update salary of Sunny with 9999?**

```

> db.employees.updateOne({ename: "Sunny"},{$set: {esal: 9999}})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }

```

```

> db.employees.find({ename: "Sunny"})
{ "_id" : 1, "eno" : 100, "ename" : "Sunny", "esal" : 1000, "eaddr" :
"Mumbai" }
> db.employees.updateOne({ename: "Sunny"},{$set: {esal: 9999}})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
> db.employees.find({ename: "Sunny"})
{ "_id" : 1, "eno" : 100, "ename" : "Sunny", "esal" : 9999, "eaddr" :
"Mumbai" }

```

## **Q2. Update all Mumbai based employees salary as 7777?**

```
> db.employees.updateOne({eaddr: "Mumbai"},{$set: {esal: 7777}})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
```

**It will perform updation only for first matched document.**

```
> db.employees.updateOne({eaddr: "Mumbai"},{$set: {esal: 7777}})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
> db.employees.find({eaddr: "Mumbai"})
{ "_id" : 1, "eno" : 100, "ename" : "Sunny", "esal" : 7777, "eaddr" :
"Mumbai" }
{ "_id" : 3, "eno" : 300, "ename" : "Chinny", "esal" : 3000, "eaddr" :
"Mumbai" }
{ "_id" : 6, "eno" : 600, "ename" : "Tinny", "esal" : 6000, "eaddr" :
"Mumbai" }
```

**Note: updateOne() will always consider only first matched document.  
If updation is not available then only it will perform updation.**

```
> db.employees.updateOne({eaddr: "Mumbai"},{$set: {esal: 7777}})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 0 }
> db.employees.find({eaddr: "Mumbai"})
{ "_id" : 1, "eno" : 100, "ename" : "Sunny", "esal" : 7777, "eaddr" :
"Mumbai" }
{ "_id" : 3, "eno" : 300, "ename" : "Chinny", "esal" : 3000, "eaddr" :
"Mumbai" }
{ "_id" : 6, "eno" : 600, "ename" : "Tinny", "esal" : 6000, "eaddr" :
"Mumbai" }
```

```
> db.employees.updateMany({eaddr: "Mumbai"},{$set: {esal: 7777}})
```

**It will update all matched documents**

```
> db.employees.find({eaddr: "Mumbai"})
{ "_id" : 1, "eno" : 100, "ename" : "Sunny", "esal" : 7777, "eaddr" :
"Mumbai" }
{ "_id" : 3, "eno" : 300, "ename" : "Chinny", "esal" : 3000, "eaddr" :
"Mumbai" }
{ "_id" : 6, "eno" : 600, "ename" : "Tinny", "esal" : 6000, "eaddr" :
"Mumbai" }
> db.employees.updateMany({eaddr: "Mumbai"},{$set: {esal: 7777}})
{ "acknowledged" : true, "matchedCount" : 3, "modifiedCount" : 2 }
> db.employees.find({eaddr: "Mumbai"})
{ "_id" : 1, "eno" : 100, "ename" : "Sunny", "esal" : 7777, "eaddr" :
"Mumbai" }
{ "_id" : 3, "eno" : 300, "ename" : "Chinny", "esal" : 7777, "eaddr" :
"Mumbai" }
{ "_id" : 6, "eno" : 600, "ename" : "Tinny", "esal" : 7777, "eaddr" :
"Mumbai" }
```

**Q3. Update all Delhi based Employees salary as 5555?**

```
> db.employees.update({eaddr: "Delhi"},{$set: {esal:5555}})
```

**It will perform updation for only first matched document.It is exactly same as updateOne() method.**

```
> db.employees.find({eaddr: "Delhi"})
{ "_id" : 4, "eno" : 400, "ename" : "Vinny", "esal" : 4000, "eaddr" :
"Delhi" }
{ "_id" : 7, "eno" : 700, "ename" : "Zinny", "esal" : 7000, "eaddr" :
"Delhi" }
> db.employees.update({eaddr: "Delhi"},{$set: {esal:5555}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

```
> db.employees.find({eaddr: "Delhi"})
{ "_id" : 4, "eno" : 400, "ename" : "Vinny", "esal" : 5555, "eaddr" :
"Delhi" }
{ "_id" : 7, "eno" : 700, "ename" : "Zinny", "esal" : 7000, "eaddr" :
"Delhi" }
```

```
> db.employees.update({eaddr: "Delhi"},{$set: {esal:5555}},{multi: true})
It will perform updation for all matched documents.
```

```
> db.employees.find({eaddr: "Delhi"})
{ "_id" : 4, "eno" : 400, "ename" : "Vinny", "esal" : 5555, "eaddr" :
"Delhi" }
{ "_id" : 7, "eno" : 700, "ename" : "Zinny", "esal" : 7000, "eaddr" :
"Delhi" }
> db.employees.update({eaddr: "Delhi"},{$set: {esal:5555}},{multi: true})
WriteResult({ "nMatched" : 2, "nUpserted" : 0, "nModified" : 1 })
> db.employees.update({eaddr: "Delhi"},{$set: {esal:4444}},{multi: true})
WriteResult({ "nMatched" : 2, "nUpserted" : 0, "nModified" : 2 })
```

### **Update Operators:**

-----

**We can use update operators to perform required updations.**

- 1. \$set**
  - 2. \$unset**
  - 3. \$rename**
  - 4. \$inc**
  - 5. \$min**
  - 6. \$max**
  - 7. \$mul**
- etc**

## **1. \$set operator:**

-----

**We can use \$set operator to set the value to the field in matched document.**

```
> db.employees.update({ename:"Sunny"},{$set: {esal:9999}})
```

### **case-1:**

-----

**If the specified field does not exist, \$set will add a new field with provided value.**

```
> db.employees.update({ename:"Sunny"},{$set: {husband: "Daniel"}})
```

```
> db.employees.find({ename:"Sunny"})
```

```
{ "_id" : 1, "eno" : 100, "ename" : "Sunny", "esal" : 7777, "eaddr" :  
"Mumbai" }
```

```
> db.employees.update({ename:"Sunny"},{$set: {husband: "Daniel"}})
```

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

```
> db.employees.find({ename:"Sunny"})
```

```
{ "_id" : 1, "eno" : 100, "ename" : "Sunny", "esal" : 7777, "eaddr" :  
"Mumbai", "husband" : "Daniel" }
```

### **Case-2: updating multiple fields at a time**

-----

**We can perform updations for multiple fields at a time.**

```
{ $set: {field1: value1, field2:value2, ...} }
```

```
> db.employees.update({ename:"Sunny"},{$set: {esal: 1111, age: 45,  
origin: "Punjab"}})
```

```

> db.employees.find({ename:"Sunny"})
{ "_id" : 1, "eno" : 100, "ename" : "Sunny", "esal" : 7777, "eaddr" :
"Mumbai", "husband" : "Daniel" }
> db.employees.update({ename:"Sunny"},{$set: {esal: 1111, age: 45,
origin: "Punjab"}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.employees.find({ename:"Sunny"})
{ "_id" : 1, "eno" : 100, "ename" : "Sunny", "esal" : 1111, "eaddr" :
"Mumbai", "husband" : "Daniel", "age" : 45, "origin" : "Punjab" }

```

**Q3. Add a new field named with friend with value Guest where esal value is >= 4000?**

```

> db.employees.updateMany({esal:{$gte: 4000}},{$set: {friend:"Guest"}})
> db.employees.update({esal:{$gte: 4000}},{$set:
{friend:"Guest"}},{multi:true})

```

```

> db.employees.find({esal:{$gte: 4000}})
{ "_id" : 3, "eno" : 300, "ename" : "Chinny", "esal" : 7777, "eaddr" :
"Mumbai" }
{ "_id" : 4, "eno" : 400, "ename" : "Vinny", "esal" : 4444, "eaddr" :
"Delhi" }
{ "_id" : 5, "eno" : 500, "ename" : "Pinny", "esal" : 5000, "eaddr" :
"Chennai" }
{ "_id" : 6, "eno" : 600, "ename" : "Tinny", "esal" : 7777, "eaddr" :
"Mumbai" }
{ "_id" : 7, "eno" : 700, "ename" : "Zinny", "esal" : 4444, "eaddr" :
"Delhi" }
> db.employees.updateMany({esal:{$gte: 4000}},{$set: {friend:"Guest"}})
{ "acknowledged" : true, "matchedCount" : 5, "modifiedCount" : 5 }

```



```

> db.employees.find({esal:{$gte: 4000}})
{ "_id" : 3, "eno" : 300, "ename" : "Chinny", "esal" : 7777, "eaddr" :
"Mumbai", "friend" : "Guest" }
{ "_id" : 4, "eno" : 400, "ename" : "Vinny", "esal" : 4444, "eaddr" :
"Delhi", "friend" : "Guest" }
{ "_id" : 5, "eno" : 500, "ename" : "Pinny", "esal" : 5000, "eaddr" :
"Chennai", "friend" : "Guest" }
{ "_id" : 6, "eno" : 600, "ename" : "Tinny", "esal" : 7777, "eaddr" :
"Mumbai", "friend" : "Guest" }
{ "_id" : 7, "eno" : 700, "ename" : "Zinny", "esal" : 4444, "eaddr" :
"Delhi", "friend" : "Guest" }
> db.employees.find()
{ "_id" : 1, "eno" : 100, "ename" : "Sunny", "esal" : 1111, "eaddr" :
"Mumbai", "husband" : "Daniel", "age" : 45, "origin" : "Punjab" }
{ "_id" : 2, "eno" : 200, "ename" : "Bunny", "esal" : 2000, "eaddr" :
"Hyderabad" }
{ "_id" : 3, "eno" : 300, "ename" : "Chinny", "esal" : 7777, "eaddr" :
"Mumbai", "friend" : "Guest" }
{ "_id" : 4, "eno" : 400, "ename" : "Vinny", "esal" : 4444, "eaddr" :
"Delhi", "friend" : "Guest" }
{ "_id" : 5, "eno" : 500, "ename" : "Pinny", "esal" : 5000, "eaddr" :
"Chennai", "friend" : "Guest" }
{ "_id" : 6, "eno" : 600, "ename" : "Tinny", "esal" : 7777, "eaddr" :
"Mumbai", "friend" : "Guest" }
{ "_id" : 7, "eno" : 700, "ename" : "Zinny", "esal" : 4444, "eaddr" :
"Delhi", "friend" : "Guest" }

```

### case-3: Nested document updation:

-----

```

{
  "_id" : ObjectId("5fe95428fe935cdac43627cf"),
  "title" : "Oracle in simple way",
  "isbn" : 6677,

```

```

    "downloadable" : true,
    "no_of_reviews" : 3,
    "tags" : [
        "database",
        "sql",
        "relational"
    ],
    "languages" : [
        "english",
        "hindi",
        "telugu"
    ],
    "author" : {
        "name" : "Virat Kohli",
        "callname" : "kohli",
        "profile" : {
            "exp" : 2,
            "courses" : 2,
            "books" : 2
        }
    }
}

```

**Q. Change call name of Virat Kohli as Virushka?**

```

> db.books.update({"author.name": "Virat Kohli"},{$set:
{"author.callname": "Virushka"}})

```

```

> db.books.find({"author.name": "Virat Kohli"}).pretty()
{
  "_id" : ObjectId("5fe95428fe935cdac43627cf"),
  "title" : "Oracle in simple way",
  "isbn" : 6677,

```

```

    "downloadable" : true,
    "no_of_reviews" : 3,
    "tags" : [
        "database",
        "sql",
        "relational"
    ],
    "languages" : [
        "english",
        "hindi",
        "telugu"
    ],
    "author" : {
        "name" : "Virat Kohli",
        "callname" : "kohli",
        "profile" : {
            "exp" : 2,
            "courses" : 2,
            "books" : 2
        }
    }
}
}
> db.books.update({"author.name": "Virat Kohli"},{$set:
{"author.callname": "Virushka"}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.books.find({"author.name": "Virat Kohli"}).pretty()
{
  "_id" : ObjectId("5fe95428fe935cdac43627cf"),
  "title" : "Oracle in simple way",
  "isbn" : 6677,
  "downloadable" : true,
  "no_of_reviews" : 3,
  "tags" : [

```

```

        "database",
        "sql",
        "relational"
    ],
    "languages" : [
        "english",
        "hindi",
        "telugu"
    ],
    "author" : {
        "name" : "Virat Kohli",
        "callname" : "Virushka",
        "profile" : {
            "exp" : 2,
            "courses" : 2,
            "books" : 2
        }
    }
}

```

## 2. \$unset operator:

-----

To delete the specified field.

### Syntax:

```
{ $unset: {field1:"",field2:"",...}}
```

The specified value in the \$unset expression (ie "") does not impact operation.

**Q1. Delete esal and husband fields where ename is "Sunny"?**

```
> db.employees.update({ename: "Sunny"},{$unset: {esal:0,husband:""}})
```

```

> db.employees.find({ename: "Sunny"})
{ "_id" : 1, "eno" : 100, "ename" : "Sunny", "esal" : 1000, "eaddr" :
"Mumbai", "husband" : "Daniel" }
> db.employees.update({ename: "Sunny"},{$unset: {esal:0,husband: ""}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.employees.find({ename: "Sunny"})
{ "_id" : 1, "eno" : 100, "ename" : "Sunny", "eaddr" : "Mumbai" }

```

**Note:** If the specified field is not available, then \$unset operator won't do anything.

```

> db.employees.update({ename: "Sunny"},{$unset: {age:0}})

> db.employees.find({ename: "Sunny"})
{ "_id" : 1, "eno" : 100, "ename" : "Sunny", "eaddr" : "Mumbai" }
> db.employees.update({ename: "Sunny"},{$unset: {age:0}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 0 })
> db.employees.find({ename: "Sunny"})
{ "_id" : 1, "eno" : 100, "ename" : "Sunny", "eaddr" : "Mumbai" }

```

**Q. Remove fields husband and friend where esal is less than 8000?**

```

> db.employees.updateMany({esal:{$lt:8000}}, {$unset:
{husband:"",friend: ""}})

```

### **3. \$rename operator:**

-----

We can use \$rename operator to rename fields, ie to change name of the field.

#### **Syntax:**

```

{$rename: {field1:<newName1>, field2:<newName2>, ...} }

```

**Q. Write Query to rename esal as salary and eaddr as city in employees collection?**

```
> db.employees.updateMany({},{$rename: {esal:"salary", eaddr: "city"}})
```

```
> db.employees.find()
```

```
{ "_id" : 1, "eno" : 100, "ename" : "Sunny", "esal" : 1000, "eaddr" :  
"Mumbai" }
```

```
{ "_id" : 2, "eno" : 200, "ename" : "Bunny", "esal" : 2000, "eaddr" :  
"Hyderabad" }
```

```
{ "_id" : 3, "eno" : 300, "ename" : "Chinny", "esal" : 3000, "eaddr" :  
"Mumbai" }
```

```
{ "_id" : 4, "eno" : 400, "ename" : "Vinny", "esal" : 4000, "eaddr" :  
"Delhi" }
```

```
{ "_id" : 5, "eno" : 500, "ename" : "Pinny", "esal" : 5000, "eaddr" :  
"Chennai" }
```

```
{ "_id" : 6, "eno" : 600, "ename" : "Tinny", "esal" : 6000, "eaddr" :  
"Mumbai" }
```

```
{ "_id" : 7, "eno" : 700, "ename" : "Zinny", "esal" : 7000, "eaddr" :  
"Delhi" }
```

```
> db.employees.updateMany({},{$rename: {esal:"salary", eaddr: "city"}})
```

```
{ "acknowledged" : true, "matchedCount" : 7, "modifiedCount" : 7 }
```

```
> db.employees.find()
```

```
{ "_id" : 1, "eno" : 100, "ename" : "Sunny", "city" : "Mumbai", "salary" :  
1000 }
```

```
{ "_id" : 2, "eno" : 200, "ename" : "Bunny", "city" : "Hyderabad", "salary"  
: 2000 }
```

```
{ "_id" : 3, "eno" : 300, "ename" : "Chinny", "city" : "Mumbai", "salary" :  
3000 }
```

```
{ "_id" : 4, "eno" : 400, "ename" : "Vinny", "city" : "Delhi", "salary" :  
4000 }
```

```
{ "_id" : 5, "eno" : 500, "ename" : "Pinny", "city" : "Chennai", "salary" :  
5000 }  
{ "_id" : 6, "eno" : 600, "ename" : "Tinny", "city" : "Mumbai", "salary" :  
6000 }  
{ "_id" : 7, "eno" : 700, "ename" : "Zinny", "city" : "Delhi", "salary" :  
7000 }
```

**Note:**

-----

**1. The \$rename operator internally performs \$unset of both old name and new name and then performs \$set with new name. Hence it won't preserve order of fields.**

**2. If the document already has a field with newName then \$rename operator removes that field and renames specified field with newName.**

**eg:**

**>**

```
db.employees.insert({_id:8,eno:800,esal:8000,eaddr:"Hyderabad",city:"  
Mumbai"})
```

```
> db.employees.find()
```

**Q. rename eaddr as city?**

```
> db.employees.update({_id:8},{ $rename: {eaddr:"city"}})
```

```
> db.employees.find({_id:8})
```

```
{ "_id" : 8, "eno" : 800, "esal" : 8000, "eaddr" : "Hyderabad", "city" :  
"Mumbai" }
```

```
> db.employees.update({_id:8},{ $rename: {eaddr:"city"}})
```

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

```
> db.employees.find({_id:8})
```

```
{ "_id" : 8, "eno" : 800, "esal" : 8000, "city" : "Hyderabad" }
```

**3. If the field to rename does not exist in the document then \$rename won't do anything.**

```
> db.employees.update({_id:8},{ $rename: {age:"totalage"}})
```

```
> db.employees.find({_id:8})
```

```
{ "_id" : 8, "eno" : 800, "esal" : 8000, "city" : "Hyderabad" }
```

```
> db.employees.update({_id:8},{ $rename: {age:"totalage"}})
```

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

```
> db.employees.find({_id:8})
```

```
{ "_id" : 8, "eno" : 800, "esal" : 8000, "city" : "Hyderabad" }
```

#### **4. \$inc operator:**

-----

**inc means increment.**

**We can use \$inc to increment or decrement value of the field with specified amount.**

```
salary=salary+1000
```

```
salary=salary-1000
```

#### **Syntax:**

```
{ $inc: {field1:amount1,field2:amount,..}}
```

**\$inc can take both positive and negative values.**

**positive value for increment operation**

**negative value for decrement operation**

#### **case study:**

-----



```

db.employees.insert({_id:1,eno:100,ename:"Sunny",esal:1000,eaddr:"M
umbai"})
db.employees.insert({_id:2,eno:200,ename:"Bunny",esal:2000,eaddr:"H
yderabad"})
db.employees.insert({_id:3,eno:300,ename:"Chinny",esal:3000,eaddr:"M
umbai"})
db.employees.insert({_id:4,eno:400,ename:"Vinny",esal:4000,eaddr:"De
lhi"})
db.employees.insert({_id:5,eno:500,ename:"Pinny",esal:5000,eaddr:"Ch
ennai"})
db.employees.insert({_id:6,eno:600,ename:"Tinny",esal:6000,eaddr:"Mu
mbai"})
db.employees.insert({_id:7,eno:700,ename:"Zinny",esal:7000,eaddr:"De
lhi"})

```

**Q1. Increment all employee salary by 500.**

```
> db.employees.updateMany({},{$inc: {esal:500}})
```

```

> db.employees.updateMany({},{$inc: {esal:500}})
{ "acknowledged" : true, "matchedCount" : 7, "modifiedCount" : 7 }
> db.employees.find()
{ "_id" : 1, "eno" : 100, "ename" : "Sunny", "esal" : 1500, "eaddr" :
"Mumbai" }
{ "_id" : 2, "eno" : 200, "ename" : "Bunny", "esal" : 2500, "eaddr" :
"Hyderabad" }
{ "_id" : 3, "eno" : 300, "ename" : "Chinny", "esal" : 3500, "eaddr" :
"Mumbai" }
{ "_id" : 4, "eno" : 400, "ename" : "Vinny", "esal" : 4500, "eaddr" :
"Delhi" }
{ "_id" : 5, "eno" : 500, "ename" : "Pinny", "esal" : 5500, "eaddr" :
"Chennai" }

```

```
{ "_id" : 6, "eno" : 600, "ename" : "Tinny", "esal" : 6500, "eaddr" :  
"Mumbai" }  
{ "_id" : 7, "eno" : 700, "ename" : "Zinny", "esal" : 7500, "eaddr" :  
"Delhi" }
```

**Q2. Decrement Employee salary by Rs 1 where esal is > 4700?**

```
> db.employees.updateMany({esal:{$gt: 4700}},{$inc: {esal:-1}})  
  
> db.employees.updateMany({esal:{$gt: 4700}},{$inc: {esal:-1}})  
{ "acknowledged" : true, "matchedCount" : 3, "modifiedCount" : 3 }  
> db.employees.find()  
{ "_id" : 1, "eno" : 100, "ename" : "Sunny", "esal" : 1500, "eaddr" :  
"Mumbai" }  
{ "_id" : 2, "eno" : 200, "ename" : "Bunny", "esal" : 2500, "eaddr" :  
"Hyderabad" }  
{ "_id" : 3, "eno" : 300, "ename" : "Chinny", "esal" : 3500, "eaddr" :  
"Mumbai" }  
{ "_id" : 4, "eno" : 400, "ename" : "Vinny", "esal" : 4500, "eaddr" :  
"Delhi" }  
{ "_id" : 5, "eno" : 500, "ename" : "Pinny", "esal" : 5499, "eaddr" :  
"Chennai" }  
{ "_id" : 6, "eno" : 600, "ename" : "Tinny", "esal" : 6499, "eaddr" :  
"Mumbai" }  
{ "_id" : 7, "eno" : 700, "ename" : "Zinny", "esal" : 7499, "eaddr" :  
"Delhi" }
```

**Note:**

**1. If the specified field does not exist, \$inc creates that field and sets that field to the specified value.**

```
> db.employees.updateMany({},{$inc: {age:2}})
```

```

{ "acknowledged" : true, "matchedCount" : 7, "modifiedCount" : 7 }
> db.employees.find()
{ "_id" : 1, "eno" : 100, "ename" : "Sunny", "esal" : 1500, "eaddr" :
"Mumbai", "age" : 2 }
{ "_id" : 2, "eno" : 200, "ename" : "Bunny", "esal" : 2500, "eaddr" :
"Hyderabad", "age" : 2 }
{ "_id" : 3, "eno" : 300, "ename" : "Chinny", "esal" : 3500, "eaddr" :
"Mumbai", "age" : 2 }
{ "_id" : 4, "eno" : 400, "ename" : "Vinny", "esal" : 4500, "eaddr" :
"Delhi", "age" : 2 }
{ "_id" : 5, "eno" : 500, "ename" : "Pinny", "esal" : 5499, "eaddr" :
"Chennai", "age" : 2 }
{ "_id" : 6, "eno" : 600, "ename" : "Tinny", "esal" : 6499, "eaddr" :
"Mumbai", "age" : 2 }
{ "_id" : 7, "eno" : 700, "ename" : "Zinny", "esal" : 7499, "eaddr" :
"Delhi", "age" : 2 }

```

**2. We cannot perform multiple updates on the same field at a time, otherwise we will get error.**

```

> db.employees.updateMany({},{$inc:{esal:500}, $set:{esal:5000}})
"errmsg" : "Updating the path 'esal' would create a conflict at 'esal'"

```

**\$set--->To set a new value to the specified field/To create a new field with provided value**

**\$unset--->To delete specified field**

**\$rename--->To rename the specified field**

**\$inc--->To perform increment and decrement operations on the field value**

**Basic understanding purpose:**

```

-----
> db.employees.find()
{ "_id" : 1, "eno" : 100, "ename" : "Sunny", "esal" : 1500, "eaddr" :
"Mumbai", "age" : 2 }
{ "_id" : 2, "eno" : 200, "ename" : "Bunny", "esal" : 2500, "eaddr" :
"Hyderabad", "age" : 2 }
{ "_id" : 3, "eno" : 300, "ename" : "Chinny", "esal" : 3500, "eaddr" :
"Mumbai", "age" : 2 }
{ "_id" : 4, "eno" : 400, "ename" : "Vinny", "esal" : 4500, "eaddr" :
"Delhi", "age" : 2 }
{ "_id" : 5, "eno" : 500, "ename" : "Pinny", "esal" : 5499, "eaddr" :
"Chennai", "age" : 2 }
{ "_id" : 6, "eno" : 600, "ename" : "Tinny", "esal" : 6499, "eaddr" :
"Mumbai", "age" : 2 }
{ "_id" : 7, "eno" : 700, "ename" : "Zinny", "esal" : 7499, "eaddr" :
"Delhi", "age" : 2 }

```

**Q1. update 5000 as minimum salary for every employee--->\$max operator**

**Q2. update 5000 as maximum salary for every employee--->\$min operator**

**Q3. Increment Every employee salary by 10% --->\$mul operator  
esal\*1.1**

**Q4. Double Every employee salary as Covid Offer --->\$mul operator  
esal\*2**

**4. \$min operator:**

**It only updates field value if the specified value is less than current field value .**

**minimum value of(provided value,current value)**

**Consider only the value which is minimum among given and current**

**Syntax: {\$min: {field1:value1, field2:value2,...}}**

**Q. To make maximum salary of every employee as 5000. If any employee salary greater than 5000 then assign 5000?**

```
> db.employees.updateMany({},{$min: {esal:5000}})
{ "acknowledged" : true, "matchedCount" : 7, "modifiedCount" : 3 }
```

```
> db.employees.find()
{ "_id" : 1, "eno" : 100, "ename" : "Sunny", "esal" : 1500, "eaddr" :
"Mumbai", "age" : 2 }
{ "_id" : 2, "eno" : 200, "ename" : "Bunny", "esal" : 2500, "eaddr" :
"Hyderabad", "age" : 2 }
{ "_id" : 3, "eno" : 300, "ename" : "Chinny", "esal" : 3500, "eaddr" :
"Mumbai", "age" : 2 }
{ "_id" : 4, "eno" : 400, "ename" : "Vinny", "esal" : 4500, "eaddr" :
"Delhi", "age" : 2 }
{ "_id" : 5, "eno" : 500, "ename" : "Pinny", "esal" : 5000, "eaddr" :
"Chennai", "age" : 2 }
{ "_id" : 6, "eno" : 600, "ename" : "Tinny", "esal" : 5000, "eaddr" :
"Mumbai", "age" : 2 }
{ "_id" : 7, "eno" : 700, "ename" : "Zinny", "esal" : 5000, "eaddr" :
"Delhi", "age" : 2 }
```

**Note: If the specified field does not exist, then \$min operator creates that field and assign with provided value.**

```
> db.employees.updateMany({},{$min:{marks: 99}})
{ "acknowledged" : true, "matchedCount" : 7, "modifiedCount" : 7 }
```

**6. \$max operator:**

-----

**\$max operator updates the value of the field to the specified value iff specified value is greater than current value.**

**i.e maximum of (provided value,current value)**

**Syntax: {\$max: {field1:value1, field2:value2,...}}**

**Q. Make minimum salary of every employee as 4000. i.e if any employee salary is less than 4000 then assign 4000?**

**max of (4000, current value)**

**4000,4500-->4500**

**4000,2300--->4000**

**> db.employees.updateMany({},{\$max: {esal:4000}})**

**if 4000 is greater than current value then only updation will be happend.**

**> db.employees.updateMany({},{\$max: {esal:4000}})**

**{ "acknowledged" : true, "matchedCount" : 7, "modifiedCount" : 3 }**

**> db.employees.find()**

**{ "\_id" : 1, "eno" : 100, "ename" : "Sunny", "esal" : 4000, "eaddr" : "Mumbai", "age" : 2 }**

**{ "\_id" : 2, "eno" : 200, "ename" : "Bunny", "esal" : 4000, "eaddr" : "Hyderabad", "age" : 2 }**

**{ "\_id" : 3, "eno" : 300, "ename" : "Chinny", "esal" : 4000, "eaddr" : "Mumbai", "age" : 2 }**

**{ "\_id" : 4, "eno" : 400, "ename" : "Vinny", "esal" : 4500, "eaddr" : "Delhi", "age" : 2 }**

**{ "\_id" : 5, "eno" : 500, "ename" : "Pinny", "esal" : 5000, "eaddr" : "Chennai", "age" : 2 }**

```
{ "_id" : 6, "eno" : 600, "ename" : "Tinny", "esal" : 5000, "eaddr" :  
"Mumbai", "age" : 2 }  
{ "_id" : 7, "eno" : 700, "ename" : "Zinny", "esal" : 5000, "eaddr" :  
"Delhi", "age" : 2 }
```

**Note:** If the specified field does not exist, then \$max operator creates that field and assign with provided value.

```
> db.employees.updateMany({},{$max:{height:5.8}})
```

```
> db.employees.updateMany({},{$max:{height:5.8}})  
{ "acknowledged" : true, "matchedCount" : 7, "modifiedCount" : 7 }  
> db.employees.find()  
{ "_id" : 1, "eno" : 100, "ename" : "Sunny", "esal" : 4000, "eaddr" :  
"Mumbai", "age" : 2, "marks" : 99, "height" : 5.8 }  
{ "_id" : 2, "eno" : 200, "ename" : "Bunny", "esal" : 4000, "eaddr" :  
"Hyderabad", "age" : 2, "marks" : 99, "height" : 5.8 }  
{ "_id" : 3, "eno" : 300, "ename" : "Chinny", "esal" : 4000, "eaddr" :  
"Mumbai", "age" : 2, "marks" : 99, "height" : 5.8 }  
{ "_id" : 4, "eno" : 400, "ename" : "Vinny", "esal" : 4500, "eaddr" :  
"Delhi", "age" : 2, "marks" : 99, "height" : 5.8 }  
{ "_id" : 5, "eno" : 500, "ename" : "Pinny", "esal" : 5000, "eaddr" :  
"Chennai", "age" : 2, "marks" : 99, "height" : 5.8 }  
{ "_id" : 6, "eno" : 600, "ename" : "Tinny", "esal" : 5000, "eaddr" :  
"Mumbai", "age" : 2, "marks" : 99, "height" : 5.8 }  
{ "_id" : 7, "eno" : 700, "ename" : "Zinny", "esal" : 5000, "eaddr" :  
"Delhi", "age" : 2, "marks" : 99, "height" : 5.8 }
```

**Note:**

1. If provided value is less than current value then only perform updation-->min operator.
2. If provided value is greater than current value then only perform updation-->max operator.

**Q. what if we don't want min/max operator to create the new field?  
Select all documents where specified field exists and then perform  
upadation.**

```
> db.employees.update({age:{$exists:true}},{$set:{age:15}},{multi:true})  
WriteResult({ "nMatched" : 7, "nUpserted" : 0, "nModified" : 7 })
```

**>**

```
db.employees.update({phone_number:{$exists:true}},{$set:{phone_num  
ber:1234}},{multi:true})  
WriteResult({ "nMatched" : 0, "nUpserted" : 0, "nModified" : 0 })
```

**phone\_number field not created.**

## **7. \$mul operator:**

**-----**

**mul means multiplication**

**We can use \$mul operator to multiply the value of a field by a number.**

```
{$mul: {field: number}}
```

**The field to update must contain numeric value.**

**Q1. Double all employee salary where esal is less than 4900?**

```
> db.employees.updateMany({esal: {$lt: 4900}},{$mul: {esal: 2}})  
{ "acknowledged" : true, "matchedCount" : 4, "modifiedCount" : 4 }
```

**Q2. Increment Salary by 10% for all employees belongs to Mumbai?**

```
> db.employees.updateMany({eaddr: "Mumbai"},{$mul: {esal: 1.1}})  
{ "acknowledged" : true, "matchedCount" : 3, "modifiedCount" : 3 }
```

**1000--->1100**



**10000--->11000**

**Note:**

**If the specified field is not available then \$mul creates that field and sets the value to zero.**

```
> db.employees.updateMany({},{$mul: {xyz: 3,abc: 3.5}})
{ "acknowledged" : true, "matchedCount" : 8, "modifiedCount" : 8 }
> db.employees.find()
...
{ "_id" : 10, "ename" : "Durga", "eaddr" : "Hyderabad", "eno" : 1000,
"esal" : 10000, "abc" : 0, "xyz" : 0 }
```

**Note:**

- 1. \$set --->To set a new value to the existing field or to create new field**
- 2. \$unset--->To unset/delete existing field**
- 3. \$rename-->To rename the value of the field.**
- 4. \$inc --->To increment or decrement field value**
- 5. \$min --->To update only if the provided value is less than current value**
- 6. \$max --->To update only if the provided value is greater than current value**
- 7. \$mul -->To multiply field value by a number**

**Understanding upsert property:**

-----

**Whenever we are trying to perform update operation, the matched document may or may not be available. If it is available then it will be updated and if it is not available then update won't happen.**

**If the document is not available then we can insert that document in the database automatically. For this we have to use upsert property.**

**upsert = update + insert**

**at the time two works update and insert. First update and if it is not possible then insert.**

**upsert will take boolean value.**

**If it is set to true, it will creates a new document if it is not available.**

**If it is set to false, then it will perform just update operation and won't create any new document.**

**The default value of upsert is: false.**

**Demo Execution:**

-----

**>db.employees.find()**

**> db.employees.find()**

```
{ "_id" : 1, "eno" : 100, "ename" : "Sunny", "esal" : 8800, "eaddr" :  
"Mumbai", "age" : 15, "marks" : 99, "height" : 5.8, "abc" : 0, "xyz" : 0 }  
{ "_id" : 2, "eno" : 200, "ename" : "Bunny", "esal" : 8000, "eaddr" :  
"Hyderabad", "age" : 15, "marks" : 99, "height" : 5.8, "abc" : 0, "xyz" : 0  
}  
{ "_id" : 3, "eno" : 300, "ename" : "Chinny", "esal" : 8800, "eaddr" :  
"Mumbai", "age" : 15, "marks" : 99, "height" : 5.8, "abc" : 0, "xyz" : 0 }  
{ "_id" : 4, "eno" : 400, "ename" : "Vinny", "esal" : 9000, "eaddr" :  
"Delhi", "age" : 15, "marks" : 99, "height" : 5.8, "abc" : 0, "xyz" : 0 }  
{ "_id" : 5, "eno" : 500, "ename" : "Pinny", "esal" : 5000, "eaddr" :  
"Chennai", "age" : 15, "marks" : 99, "height" : 5.8, "abc" : 0, "xyz" : 0 }  
{ "_id" : 6, "eno" : 600, "ename" : "Tinny", "esal" : 5500, "eaddr" :  
"Mumbai", "age" : 15, "marks" : 99, "height" : 5.8, "abc" : 0, "xyz" : 0 }  
{ "_id" : 7, "eno" : 700, "ename" : "Zinny", "esal" : 5000, "eaddr" :  
"Delhi", "age" : 15, "marks" : 99, "height" : 5.8, "abc" : 0, "xyz" : 0 }  
{ "_id" : 10, "ename" : "Durga", "eaddr" : "Hyderabad", "eno" : 1000,  
"esal" : 10000, "abc" : 0, "xyz" : 0 }
```

```
> db.employees.update({ename: "Mallika"}, {$set:
{_id:11,esal:9999,eaddr:"Mumabi"}})
WriteResult({ "nMatched" : 0, "nUpserted" : 0, "nModified" : 0 })
```

```
> db.employees.update({ename: "Mallika"}, {$set:
{_id:11,esal:9999,eaddr:"Mumabi"}}, {upsert: false})
WriteResult({ "nMatched" : 0, "nUpserted" : 0, "nModified" : 0 })
```

```
> db.employees.update({ename: "Mallika"}, {$set:
{_id:11,esal:9999,eaddr:"Mumabi"}}, {upsert: true})
WriteResult({ "nMatched" : 0, "nUpserted" : 1, "nModified" : 0, "_id" : 11
})
```

```
> db.employees.find()
...
{ "_id" : 11, "ename" : "Mallika", "eaddr" : "Mumabi", "esal" : 9999 }
```

### **Array Update Operators:**

-----

**1. \$ 2. \$[] 3. \$[<identifier>]**

#### **1. Updating First Matched element by using \$:**

-----

**\$ acts as a placeholder to update first matched element based on query condition.**

#### **Syntax:**

```
db.collection.update(query,{update_operator:{"<array>.$" : value}})
```

**The array field must appear as the part of query condition.**

## Case Study:

-----

```
db.students.insertOne({_id:1, marks: [70,87,90,30,40]})
db.students.insertOne({_id:2, marks: [90,88,92,110,45]})
db.students.insertOne({_id:3, marks: [85,100,90,76,58]})
db.students.insertOne({_id:4, marks: [79,85,80,89,56]})
db.students.insertOne({_id:5, marks: [88,88,92,45,23]})
db.students.insertOne({_id:6, marks: [95,90,96,92,95]})
```

**Q1. Update the first matched element 90 in marks array to 999 where \_id:1?**

```
> db.students.update({_id:1,marks:90},{ $set: {"marks.$": 999}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

```
> db.students.find()
{ "_id" : 1, "marks" : [ 70, 87, 90, 30, 40 ] }
{ "_id" : 2, "marks" : [ 90, 88, 92, 110, 45 ] }
{ "_id" : 3, "marks" : [ 85, 100, 90, 76, 58 ] }
{ "_id" : 4, "marks" : [ 79, 85, 80, 89, 56 ] }
{ "_id" : 5, "marks" : [ 88, 88, 92, 45, 23 ] }
{ "_id" : 6, "marks" : [ 95, 90, 96, 92, 95 ] }
> db.students.update({_id:1,marks:90},{ $set: {"marks.$": 999}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.students.find()
{ "_id" : 1, "marks" : [ 70, 87, 999, 30, 40 ] }
{ "_id" : 2, "marks" : [ 90, 88, 92, 110, 45 ] }
{ "_id" : 3, "marks" : [ 85, 100, 90, 76, 58 ] }
{ "_id" : 4, "marks" : [ 79, 85, 80, 89, 56 ] }
{ "_id" : 5, "marks" : [ 88, 88, 92, 45, 23 ] }
{ "_id" : 6, "marks" : [ 95, 90, 96, 92, 95 ] }
```

**Q2. Update the first matched element in marks array which is less than 90 with 90 in every document?**

```
> db.students.updateMany({marks:{$elemMatch:{$lt:90}}},{ $set:
{"marks.$": 90}})
```

```
> db.students.find()
{ "_id" : 1, "marks" : [ 70, 87, 999, 30, 40 ] }
{ "_id" : 2, "marks" : [ 90, 88, 92, 110, 45 ] }
{ "_id" : 3, "marks" : [ 85, 100, 90, 76, 58 ] }
{ "_id" : 4, "marks" : [ 79, 85, 80, 89, 56 ] }
{ "_id" : 5, "marks" : [ 88, 88, 92, 45, 23 ] }
{ "_id" : 6, "marks" : [ 95, 90, 96, 92, 95 ] }
> db.students.updateMany({marks:{$elemMatch:{$lt:90}}},{ $set:
{"marks.$": 90}})
{ "acknowledged" : true, "matchedCount" : 5, "modifiedCount" : 5 }
> db.students.find()
{ "_id" : 1, "marks" : [ 90, 87, 999, 30, 40 ] }
{ "_id" : 2, "marks" : [ 90, 90, 92, 110, 45 ] }
{ "_id" : 3, "marks" : [ 90, 100, 90, 76, 58 ] }
{ "_id" : 4, "marks" : [ 90, 85, 80, 89, 56 ] }
{ "_id" : 5, "marks" : [ 90, 88, 92, 45, 23 ] }
{ "_id" : 6, "marks" : [ 95, 90, 96, 92, 95 ] }
```

## **2. Updating all array elements by using \$[]:**

-----  
**\$[] acts as placeholder to update all elements in the array for the matched documents based on query condition.**

**Syntax:**

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

**Q1. To increment every element of marks array by 10?**

```
> db.students.updateMany({},{$inc: {"marks.$[]": 10}})
> db.students.find()
{ "_id" : 1, "marks" : [ 90, 87, 999, 30, 40 ] }
{ "_id" : 2, "marks" : [ 90, 90, 92, 110, 45 ] }
{ "_id" : 3, "marks" : [ 90, 100, 90, 76, 58 ] }
{ "_id" : 4, "marks" : [ 90, 85, 80, 89, 56 ] }
{ "_id" : 5, "marks" : [ 90, 88, 92, 45, 23 ] }
{ "_id" : 6, "marks" : [ 95, 90, 96, 92, 95 ] }
> db.students.updateMany({},{$inc: {"marks.$[]": 10}})
{ "acknowledged" : true, "matchedCount" : 6, "modifiedCount" : 6 }
> db.students.find()
{ "_id" : 1, "marks" : [ 100, 97, 1009, 40, 50 ] }
{ "_id" : 2, "marks" : [ 100, 100, 102, 120, 55 ] }
{ "_id" : 3, "marks" : [ 100, 110, 100, 86, 68 ] }
{ "_id" : 4, "marks" : [ 100, 95, 90, 99, 66 ] }
{ "_id" : 5, "marks" : [ 100, 98, 102, 55, 33 ] }
{ "_id" : 6, "marks" : [ 105, 100, 106, 102, 105 ] }
```

**Q2. Update every element of marks array as 1000 if array contains atleast one element which is greater than or equal to 1000?**

```
> db.students.updateMany({marks:{$elemMatch:{$gte:1000}}},{ $set: {"marks.$[]": 1000}})
```

```
> db.students.find()
{ "_id" : 1, "marks" : [ 100, 97, 1009, 40, 50 ] }
{ "_id" : 2, "marks" : [ 100, 100, 102, 120, 55 ] }
{ "_id" : 3, "marks" : [ 100, 110, 100, 86, 68 ] }
```

```

{ "_id" : 4, "marks" : [ 100, 95, 90, 99, 66 ] }
{ "_id" : 5, "marks" : [ 100, 98, 102, 55, 33 ] }
{ "_id" : 6, "marks" : [ 105, 100, 106, 102, 105 ] }
> db.students.updateMany({marks:{$elemMatch:{$gte:1000}}},{ $set:
{"marks.$[]": 1000}})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
> db.students.find()
{ "_id" : 1, "marks" : [ 1000, 1000, 1000, 1000, 1000 ] }
{ "_id" : 2, "marks" : [ 100, 100, 102, 120, 55 ] }
{ "_id" : 3, "marks" : [ 100, 110, 100, 86, 68 ] }
{ "_id" : 4, "marks" : [ 100, 95, 90, 99, 66 ] }
{ "_id" : 5, "marks" : [ 100, 98, 102, 55, 33 ] }
{ "_id" : 6, "marks" : [ 105, 100, 106, 102, 105 ] }

```

### 3. Updating specific array elements by using \$[identifier]:

---

Instead of updating only first matched element or all elements of the array, we can update only required array elements. For this we have to use \$[identifier]

**\$[identifier] --->Acts as placeholder to update all elements that match the arrayFilters condition for the documents that match query condition.**

**In this case updation is based on arrayFilters condition but not based on query condition.**

**Syntax:**

**Q. Update all marks array elements which are less than 100 as 100?**

```

> db.students.updateMany({},{$set: {"marks.$[element]":
100}},{arrayFilters:[{"element": {$lt: 100}}]})

> db.students.find()
{ "_id" : 1, "marks" : [ 1000, 1000, 1000, 1000, 1000 ] }
{ "_id" : 2, "marks" : [ 100, 100, 102, 120, 55 ] }
{ "_id" : 3, "marks" : [ 100, 110, 100, 86, 68 ] }
{ "_id" : 4, "marks" : [ 100, 95, 90, 99, 66 ] }
{ "_id" : 5, "marks" : [ 100, 98, 102, 55, 33 ] }
{ "_id" : 6, "marks" : [ 105, 100, 106, 102, 105 ] }
> db.students.updateMany({},{$set: {"marks.$[element]":
100}},{arrayFilters:[{"element": {$lt: 100}}]})
{ "acknowledged" : true, "matchedCount" : 6, "modifiedCount" : 4 }
> db.students.find()
{ "_id" : 1, "marks" : [ 1000, 1000, 1000, 1000, 1000 ] }
{ "_id" : 2, "marks" : [ 100, 100, 102, 120, 100 ] }
{ "_id" : 3, "marks" : [ 100, 110, 100, 100, 100 ] }
{ "_id" : 4, "marks" : [ 100, 100, 100, 100, 100 ] }
{ "_id" : 5, "marks" : [ 100, 100, 102, 100, 100 ] }
{ "_id" : 6, "marks" : [ 105, 100, 106, 102, 105 ] }

```

**Q. If we specify both query and arrayFilters in this case what is the order?**

**Ans:**

**query condition helpful to select documents.**

**In those selected documents, based on arrayFilters condition array elements will be updated.**

**Q2. Write query to perform the following update?**

**If the marks in the range 101 to 110 make as 110.**



```
> db.students.updateMany({},{$set: {"marks.$[e1]": 110}},{arrayFilters:
[{$and: [{"e1":{$gt:100}},{"e1":{$lte:110}}]}})
```

```
> db.students.find()
```

```
{ "_id" : 1, "marks" : [ 1000, 1000, 1000, 1000, 1000 ] }
```

```
{ "_id" : 2, "marks" : [ 100, 100, 102, 120, 100 ] }
```

```
{ "_id" : 3, "marks" : [ 100, 110, 100, 100, 100 ] }
```

```
{ "_id" : 4, "marks" : [ 100, 100, 100, 100, 100 ] }
```

```
{ "_id" : 5, "marks" : [ 100, 100, 102, 100, 100 ] }
```

```
{ "_id" : 6, "marks" : [ 105, 100, 106, 102, 105 ] }
```

```
> db.students.updateMany({},{$set: {"marks.$[e1]": 110}},{arrayFilters:
[{$and: [{"e1":{$gt:100}},{"e1":{$lte:110}}]}})
```

```
{ "acknowledged" : true, "matchedCount" : 6, "modifiedCount" : 3 }
```

```
> db.students.find()
```

```
{ "_id" : 1, "marks" : [ 1000, 1000, 1000, 1000, 1000 ] }
```

```
{ "_id" : 2, "marks" : [ 100, 100, 110, 120, 100 ] }
```

```
{ "_id" : 3, "marks" : [ 100, 110, 100, 100, 100 ] }
```

```
{ "_id" : 4, "marks" : [ 100, 100, 100, 100, 100 ] }
```

```
{ "_id" : 5, "marks" : [ 100, 100, 110, 100, 100 ] }
```

```
{ "_id" : 6, "marks" : [ 110, 100, 110, 110, 110 ] }
```

**Q3. Consider the students collection:**

```
db.students.insertOne({_id:1,marks:[70,87,90]})
```

```
db.students.insertOne({_id:2,marks:[90,88,92]})
```

```
db.students.insertOne({_id:3,marks:[85,100,90]})
```

```
db.students.insertOne({_id:4,marks:[79,85,80]})
```

```
db.students.insertOne({_id:5,marks:[88,88,92]})
```

```
db.students.insertOne({_id:6,marks:[95,90,96]})
```

**Write query to perform the following update?**

**If the marks in the range 71 to 80 make as 80**

**If the marks in the range 81 to 90 make as 90**

**If the marks in the range 91 to 100 make as 100**

```
> db.students.updateMany({},{$set: {"marks.$[e1]": 80, "marks.$[e2]":  
90, "marks.$[e3]": 100}},{arrayFilters: [ {$and:  
[{"e1":{$gt:70}},{"e1":{$lte:80}}]},  {$and:  
[{"e2":{$gt:80}},{"e2":{$lte:90}}]}, {$and: :{$gt:90}},{"e3":{$lte:100}}]}}))
```

```
db.students.updateMany(  
  {},  
  {  
    $set: {  
      "marks.$[e1]": 80,  
      "marks.$[e2]": 90,  
      "marks.$[e3]": 100  
    }  
  },  
  {  
    arrayFilters: [  
      {$and: [{"e1":{$gt:70}},{"e1":{$lte:80}}]},  
      {$and: [{"e2":{$gt:80}},{"e2":{$lte:90}}]},  
      {$and: [{"e3":{$gt:90}},{"e3":{$lte:100}}]}  
    ]  
  }  
)  
{ "acknowledged" : true, "matchedCount" : 6, "modifiedCount" : 6 }  
  
> db.students.find()  
{ "_id" : 1, "marks" : [ 70, 90, 90 ] }  
{ "_id" : 2, "marks" : [ 90, 90, 100 ] }  
{ "_id" : 3, "marks" : [ 90, 100, 90 ] }  
{ "_id" : 4, "marks" : [ 80, 90, 80 ] }  
{ "_id" : 5, "marks" : [ 90, 90, 100 ] }
```

```
{ "_id" : 6, "marks" : [ 100, 90, 100 ] }
```

**Note:**

**\$ --->To update only first matched element of array**

**\$[] --->To update all elements of array**

**\$[identifier] ---> To update specific array elements**

**4. Adding elements to the array by using \$push operator:**

-----

**We can use \$push operator to add elements to the array.**

**By default element will be added at the end, but based on our requirement we can add in our required position also.**

**Syntax:**

**db.collection.update({},{\$push: {<array1>: value1,...}})**

**eg-1: Adding a single element:**

-----

```
{ "_id" : 1, "marks" : [ 70, 90, 90 ] }
```

```
> db.students.update({_id:1}, {$push: {marks: 35}})
```

```
{ "_id" : 1, "marks" : [ 70, 90, 90, 35 ] }
```

**eg-2: Adding multiple elements:**

-----

**1st way:**

-----

```
> db.students.update({_id:1}, {$push: {marks: 36,marks: 37, marks: 38}})
```

```
{ "_id" : 1, "marks" : [ 70, 90, 90, 35, 38 ] }
```

**It is not added 3 elements only one element added.**

**Reason: In Javascript object, duplicate keys are not allowed. If we are trying to add entry with duplicate key, old value will be replaced with new value.**

**{marks: 36,marks: 37, marks: 38} ==>{marks: 38}**

**2nd way:**

-----

**> db.students.update({\_id:1}, {\$push: {marks: [39,40,41]}})**

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

**> db.students.find()**

**{ "\_id" : 1, "marks" : [ 70, 90, 90, 35, 38, [ 39, 40, 41 ] ] }**

**Total array added as single element.**

**We can add elements of the array individually by using \$each modifier.**

**\$each modifier:**

-----

**We can use \$each modifier to add multiple values to the array.**

**Syntax:**

**{ \$push: { <array>: {\$each: [value1,value2,...]} }**

**> db.students.update({\_id:1}, {\$push: {marks: {\$each: [42,43,44]}}})**

**> db.students.update({\_id:1}, {\$push: {marks: {\$each: [42,43,44]}}})**

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

**> db.students.find()**

**{ "\_id" : 1, "marks" : [ 70, 90, 90, 35, 38, [ 39, 40, 41 ], 42, 43, 44 ] }**

**\$position modifier:**

-----

**By default elements will be added at the end of the array. But we can add elements in the required position. For this we have to use \$position modifier.**

**To use \$position modifier, compulsory we should use \$each modifier. i.e \$position without \$each is always invalid.**

**Syntax:**

```
{  
  $push: {  
    <array>: {  
      $each: [value1,value2,value3],  
      $position: <num>  
    }  
  }  
}
```

**<num> indicates the position where we have to add element. Array follows zero based index. ie index of first element is 0.**

**eg-1: To add element at the beginning:**

-----

**We have to use <num> as 0.**

```
> db.students.insert({_id:7,marks:[10,20,30,40]})
```

```
> db.students.update(  
  {_id:7},  
  {  
    $push: {  
      marks: {  
        $each:[50],  
        $position:0  
      }  
    }  
  })
```

```

    }
  }
)

{ "_id" : 7, "marks" : [ 50, 10, 20, 30, 40 ] }

```

**eg-2: To add element at 3rd index place:**

```

> db.students.update(
  { _id:7},
  {
    $push: {
      marks: {
        $each:[60],
        $position:3
      }
    }
  }
)

```

```

{ "_id" : 7, "marks" : [ 50, 10, 20, 60, 30, 40 ] }

```

**Negative Index:**

**We can use negative index to add elements from the end. -1 indicates the position just before last element in the array.**

**eg-1:**

```

> db.students.update(
  { _id:7},
  {
    $push: {
      marks: {

```

```

        $each:[70],
        $position:-1
    }
}
)
{ "_id" : 7, "marks" : [ 50, 10, 20, 60, 30, 70, 40 ] }

```

**eg-2:**

```

db.students.update(
    { _id:7},
    {
        $push: {
            marks: {
                $each:[80],
                $position:-2
            }
        }
    }
)
{ "_id" : 7, "marks" : [ 50, 10, 20, 60, 30, 80, 70, 40 ] }

```

**Q. IF NUM > size of array , then will it be added at the end ?**

**Yes**

**eg-3:**

```

db.students.update(
    { _id:7},
    {
        $push: {
            marks: {
                $each:[1,2,3],

```

```

        $position:-3
      }
    }
  }
)
{ "_id" : 7, "marks" : [ 50, 10, 20, 60, 30, 1, 2, 3, 80, 70, 40 ] }

```

**Note:** with a negative index position, if we specify multiple elements in the \$each array, the last added element is in the specified position from the end.

#### **\$sort modifier:**

-----

**We can use \$sort modifier to sort elements of the array while performing push operation.**

**To use \$sort modifier, we should use \$each modifier. i.e without \$each, we cannot use \$sort modifier.**

**We can pass empty array [], to \$each modifier to see effect of only \$sort.**

#### **Syntax:**

-----

```

db.collection.update(
  {},
  {
    $push: {
      <array>: { $each: [value1,value2,..],
                  $sort: 1|-1
                }
    }
  }
)

```



**}}**

**1 means ascending order**

**-1 means descending order**

**eg-1: Sorting array elements according to ascending order:**

-----  
**{ "\_id" : 7, "marks" : [ 10, 20, 60, 30, 1, 2, 3, 80, 70, 40, 897, 98, 99, 100, 34, 35, 36 ] }**

**> db.students.update({\_id:7},{ \$push: {marks: { \$each:[15,25,10], \$sort: 1}}})**

**{ "\_id" : 7, "marks" : [ 1, 2, 3, 10, 10, 15, 20, 25, 30, 34, 35, 36, 40, 60, 70, 80, 98, 99, 100, 897 ] }**

**eg-2: Sorting without adding any element:**

-----  
**> db.students.update({\_id:7},{ \$push: {marks: { \$each:[], \$sort: -1}}})**  
**{ "\_id" : 7, "marks" : [ 897, 100, 99, 98, 80, 70, 60, 40, 36, 35, 34, 30, 25, 20, 15, 10, 10, 3, 2, 1 ] }**

**\$slice modifier:**

-----  
**The \$slice modifier limits the number of array elements during \$push operation.**

**To use \$slice modifier, we should use \$each modifier. i.e without \$each, we cannot use \$slice modifier.**

**We can pass empty array [], to \$each modifier to see effect of only \$slice modifier.**

**Syntax:**

-----

```
db.collection.update(
    {},
    {
        $push: {
            <array>: { $each: [value1,value2,..],
                      $slice: <num>
                    }
        }
    })
```

**The <num> can be:**

- 1. zero --->To update array to an empty array.**
- 2. positive --->To update array field to contain only first <num> elements.**
- 3. Negative --->To update array field to contain only last <num> elements.**

**eg-1: To update array with last 6 elements**

-----

```
{ "_id" : 7, "marks" : [ 897, 100, 99, 98, 80, 70, 60, 40, 36, 35, 34, 30, 25,
20, 15, 10, 10, 3, 2, 1 ] }
```

```
> db.students.update({_id: 7},{ $push: {marks: { $each:[5,6,7], $slice: -6}}})
```

```
{ "_id" : 7, "marks" : [ 3, 2, 1, 5, 6, 7 ] }
```

**eg-2: To update array with first 3 elements:**

-----  
**> db.students.update({\_id: 7},{ \$push: {marks: { \$each:[], \$slice: 3}}})**  
**{ "\_id" : 7, "marks" : [ 3, 2, 1 ] }**

**eg-3: To update array with zero number of elements:**

-----  
**> db.students.update({\_id: 7},{ \$push: {marks: { \$each:[], \$slice: 0}}})**  
**{ "\_id" : 7, "marks" : [ ] }**

**The effect of order of modifiers:**

-----  
**Order of modifiers in the query is not important and we can take in any order.**

**But MongoDB Server will process push operation in the following order:**

- 1. Update array to add elements in the correct position.**
- 2. Apply sort, if specified.**
- 3. slice the array, if specified.**
- 4. Store the array**

**eg:**

```
> db.students.update(
  { _id:7},
  {
    $push: {marks: { $slice:3, $sort: -1, $each:[4,1,7,2,6,3,9,2,8,4,5]}}
  }
)

{ "_id" : 7, "marks" : [ 9, 8, 7 ] }
```

**Note:**

**1. If the specified array is not already available then \$push adds that array field with values as its elements.**

```
> db.students.update({_id:7},{ $push: {marks1: { $each: [10,20,30]}}})  
{ "_id" : 7, "marks" : [ 9, 8, 7 ], "marks1" : [ 10, 20, 30 ] }
```

**2. If the field is not array, then \$push operation will fail.**

```
> db.students.update({_id:7},{ $set: {name:"Durga"}})  
{ "_id" : 7, "marks" : [ 9, 8, 7 ], "marks1" : [ 10, 20, 30 ], "name" :  
"Durga" }  
> db.students.update({_id:7},{ $push: {name:{ $each: [10,20,30]}}})  
"errmsg" : "The field 'name' must be an array but is of type string in  
document {_id: 7.0}"
```

**Summary:**

-----

**\$push operator --->To add elements to array.**

**\$each modifier --->To add multiple elements**

**\$position modifier --->To add elements at specified position**

**\$sort modifier --->To sort elements after addition**

**\$slice modifier --->To limit the number of elements.**

**5. \$addToSet operator:**

-----

**It is exactly same as \$push operator except that it won't allow duplicates.**

**It adds elements to the array iff array does not contain already those elements.**

**There is no effect on already existing duplicates.**

### **case study:**

-----

```
db.students.insertOne({_id:1,marks:[70,87,90]})
db.students.insertOne({_id:2,marks:[90,88,92]})
db.students.insertOne({_id:3,marks:[85,100,90]})
db.students.insertOne({_id:4,marks:[79,85,80]})
db.students.insertOne({_id:5,marks:[88,88,92]})
db.students.insertOne({_id:6,marks:[95,90,96]})
```

### **eg-1: Adding duplicate element**

-----

```
{_id:5,marks:[88,88,92]}
> db.students.update({_id:5},{ $addToSet: {marks: 88}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 0 })
{ "_id" : 5, "marks" : [ 88, 88, 92 ] }
```

**In this case 88 won't be added because it is already available.**

### **eg-2: Adding non-duplicate element:**

-----

```
> db.students.update({_id:5},{ $addToSet: {marks: 90}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
{ "_id" : 5, "marks" : [ 88, 88, 92, 90 ] }
```

### **eg-3: Adding multiple elements:**

-----

**To add multiple elements we have to use \$each modifier.**

```
> db.students.update({_id:5},{ $addToSet: {marks: { $each:
[10,20,88,90,30]}}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

```
{ "_id" : 5, "marks" : [ 88, 88, 92, 90, 10, 20, 30 ] }
```

**Note: In the case of \$push operator, order terminology is applicable. Hence we can use \$position, \$sort, \$slice modifiers.**

**But in the case of \$addToSet operator, order terminology is not applicable. Hence we cannot use \$position, \$sort, \$slice modifiers.**

**But \$each modifier applicable for both \$push and \$addToSet operators.**

```
> db.students.update({_id:5},{ $addToSet: {marks: { $each: [7,8,9],  
$position: 2}}})
```

```
"errmsg" : "Found unexpected fields after $each in $addToSet: {  
$each: [ 7.0, 8.0, 9.0 ], $position: 2.0 }"
```

## **6. Removing Elements by using \$pop operator:**

-----

**We can use \$pop operator to remove either first or last element from the array.**

### **Syntax:**

-----

```
{  
  $pop: {<array>:-1|1}  
}
```

**-1 --->To remove the first element**

**1 --->To remove the last element**

**eg-1: To remove first element:**

```
-----  
{ "_id" : 5, "marks" : [ 88, 88, 92, 90, 10, 20, 30 ] }  
> db.students.update({_id: 5},{ $pop: {marks: -1}})  
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })  
{ "_id" : 5, "marks" : [ 88, 92, 90, 10, 20, 30 ] }
```

**eg-2: To remove last element:**

```
-----  
> db.students.update({_id: 5},{ $pop: {marks: 1}})  
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })  
{ "_id" : 5, "marks" : [ 88, 92, 90, 10, 20 ] }
```

**7. Remove elements by using \$pull operator:**

-----  
We can use \$pull operator either

1. To remove all instances of specified element.
2. To remove elements that match the given condition.

**Syntax:**

```
-----  
{  
  $pull: {<array>: <value> | <condition> }  
}
```

**eg-1: To delete all instances of 10**

```
-----  
{ "_id" : 7, "marks" : [ 10, 20, 30, 10, 20, 40, 50, 60, 70, 80, 90, 100 ] }  
> db.students.update({_id:7}, { $pull: {marks: 10}})
```

```
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })  
{ "_id" : 7, "marks" : [ 20, 30, 20, 40, 50, 60, 70, 80, 90, 100 ] }
```

**eg-2: To remove all elements which are greater than or equal to 50**

---

```
> db.students.update({_id:7}, {$pull: {marks: {$gte: 50}}})  
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })  
{ "_id" : 7, "marks" : [ 20, 30, 20, 40 ] }
```

## **8. Removing elements by using \$pullAll operator:**

---

**By using \$pull, we can delete either all instances of a single element or elements based on some condition.**

**But by using \$pullAll, we can delete all instances of given list of multiple elements.**

**Syntax:**

```
{  
  $pullAll: {<array>: [value1, value2, value3, ...]}  
}
```

**eg:**

```
> db.students.insert({_id:8,  
marks:[10,20,10,10,20,20,10,30,30,40,50,60]})  
> db.students.update({_id:8},{$pullAll: {marks: [10,20,30,40]}})  
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })  
{ "_id" : 8, "marks" : [ 50, 60 ] }
```

## **Summary of array update operators:**

---

### **1. \$**



2. **\$[]**
3. **\$[element]**
4. **\$push operators with modifiers: \$each, \$position, \$sort, \$slice**
5. **\$addToSet operator with \$each modifier**
6. **\$pop**
7. **\$pull**
8. **\$pullAll**

#### **CRUD Operations:**

**C --->Create|Insert**

**R --->Retrieve | Read**

**U --->Update**

**D --->Delete**

#### **Deleting Documents from the collection:**

-----

**MongoDB provides the following methods to delete documents from the collection.**

1. **deleteOne()**
2. **deleteMany()**
3. **remove()**

##### **1. deleteOne():**

-----

**To delete only one document that matches the query criteria.**

**> db.collection.deleteOne({query})**

##### **2. deleteMany():**

-----

**To delete all matched documents that matches query criteria.**

**> db.collection.deleteMany({query})**

### **Case Study:**

-----

**db.employees.insert({\_id:1,eno:100,ename:"Sunny",esal:1000,eaddr:"Mumbai"})**

**db.employees.insert({\_id:2,eno:200,ename:"Bunny",esal:2000,eaddr:"Hyderabad"})**

**db.employees.insert({\_id:3,eno:300,ename:"Chinny",esal:3000,eaddr:"Mumbai"})**

**db.employees.insert({\_id:4,eno:400,ename:"Vinny",esal:4000,eaddr:"Delhi"})**

**db.employees.insert({\_id:5,eno:500,ename:"Pinny",esal:5000,eaddr:"Chennai"})**

**db.employees.insert({\_id:6,eno:600,ename:"Tinny",esal:6000,eaddr:"Mumbai"})**

**db.employees.insert({\_id:7,eno:700,ename:"Zinny",esal:7000,eaddr:"Delhi"})**

**Q1. Delete the first matched document where eaddr is Mumbai?**

**> db.employees.deleteOne({eaddr: "Mumbai"})**

**{ "acknowledged" : true, "deletedCount" : 1 }**

**In this case only first matched document deleted.**

**Q2. Delete all the documents where eaddr is Mumbai?**

**> db.employees.deleteMany({eaddr: "Mumbai"})**

**{ "acknowledged" : true, "deletedCount" : 2 }**

**Q3. Delete all documents where esal is greater than 3000?**

**>db.employees.deleteMany({esal: {\$gt: 3000}})**

```
{ "acknowledged" : true, "deletedCount" : 3 }
```

**Q4. Delete all documents where esal is greater than 5000 and eaddr is Delhi?**

**1st way:**

**-----**

```
> db.employees.deleteMany({$and: [{esal: {$gt: 5000}}, {eaddr: "Delhi"}]})
```

```
{ "acknowledged" : true, "deletedCount" : 1 }
```

**2nd way:**

**-----**

```
db.employees.deleteMany({esal: {$gt: 5000}, eaddr: "Delhi"})
```

**Q5. How to delete all documents from the collection without deleting collection?**

```
> db.employees.deleteMany({})
```

```
> db.employees.find().count()
```

```
6
```

```
> db.employees.deleteMany({})
```

```
{ "acknowledged" : true, "deletedCount" : 6 }
```

```
> db.employees.find().count()
```

```
0
```

```
> show collections
```

```
employees
```

```
students
```

**This operation is exactly same as truncate operation in our relational databases.**

**3. remove() :**

-----  
**We can use remove() method to delete either a single document or multiple documents.**

**Syntax:**

**> db.collection.remove({query},justOne)**

**justOne field can take boolean value.**

**If it is true, then only one document will be deleted.**

**If it is false, then all matched documents will be deleted.**

**The default value is false. Hence , bydefault remove() method will delete multiple documents.**

**Q1. To delete all documents where eaddr is Mumbai?**

**> db.employees.remove({eaddr: "Mumbai"})**

**WriteResult({ "nRemoved" : 3 })**

**Q2. To delete only first matched document where eaddr is Delhi?**

**> db.employees.remove({eaddr: "Delhi"},true)**

**WriteResult({ "nRemoved" : 1 })**

**How to delete a collection?**

-----  
**> db.collection.drop()**

**To delete collection including all documents.**

**> show collections**

**employees**

**students**

**> db.employees.drop()**

**true**

**> show collections**

**students**

**How to delete a database?**

-----

**> db.dropDatabase()**

**It will delete the current database.**

**> show dbs**

**admin 0.000GB**

**config 0.000GB**

**durgadb 0.000GB**

**local 0.000GB**

**storedb 0.000GB**

**studentdb 0.000GB**

**> db.getName()**

**storedb**

**> db.dropDatabase()**

**{ "dropped" : "storedb", "ok" : 1 }**

**> show dbs**

**admin 0.000GB**

**config 0.000GB**

**durgadb 0.000GB**

**local 0.000GB**

**studentdb 0.000GB**

**Note: In general deleting collections and databases are responsibilities of database admins.**

**Q1. What is the difference between the following 2 commands?**

**db.collection.drop()**

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

**Ans:**

**db.collection.drop() --->Both collection and documents will be deleted.**

**db.collection.deleteMany({}) --->Only documents will be deleted but not collection.**

**Q2. What is the difference between the following commands?**

**> db.collection.deleteMany({})**

**> db.collection.remove({})**

**> db.collection.remove({},false)**

**All are equal and deleting all documents from the collection.**

**MongoDB Utilities (Database Tools):**

-----

**1. mongoimport**

**2. mongoexport**

**3. mongodump**

**4. mongorestore**

**etc**

**All these are separate applications.**

**These can be used for data management in mongodb.**

**These are command line utilities and we have to execute from command prompt only but not from mongodb shell.**

**By default these tools are not available and we have to install separately.**

**How to install these tools:**

-----

**<https://www.mongodb.com/try/download/database-tools>**

## **Download**

**we will get zip file: mongodb-database-tools-windows-x86\_64-100.3.0**

**Extract this zip file and copy the utility tools to our mongodb bin folder.**

**C:\Program Files\MongoDB\Server\4.4\bin**

### **1. mongoimport:**

-----

**We can use this tool to import data into mongodb database from the files like json file, csv file etc**

**eg-1: To import data from json file:**

-----

#### **Syntax:**

**mongoimport -d databaseName -c collectionName --file fileName --jsonArray**

**eg:**

**mongoimport -d durgadb -c students --file students.json --jsonArray**

**eg-2: To import data from csv file:**

-----

#### **Syntax:**

**mongoimport -d databaseName -c collectionName --type csv --headerline --drop emp.csv**

### **2. mongoexport:**

-----

**We can use this tool to export specific data from the given collection to the files.**

**The data will be stored in the file in json format.**

**Syntax:**

**mongoexport -d databaseName -c collectionName -o fileName**

**-d ==>databaseName**

**-c ==>collectionName**

**-o ==>Name of the file where exported data should be written.**

**eg: To export data from employees collection of durgadb database to emp.txt file**

**mongoexport -d durgadb -c employees -o emp.txt**

**C:\Users\lenovo\Desktop>mongoexport -d durgadb -c employees -o emp.txt**

**2021-02-14T09:18:07.170+0530 connected to: mongodb://localhost/**

**2021-02-14T09:18:07.178+0530 exported 8 records**

**emp.txt:**

**-----**

**{"\_id":{"\_id":"5fe220cd573f5ff261265825"},"eno":100.0,"ename":"Sunny","esal":1000.0,"eaddr":"Mumbai"}**

**{"\_id":{"\_id":"5fe221b6573f5ff261265826"},"eno":200.0,"ename":"Bunny","esal":2000.0,"eaddr":"Hyd"}**

**{"\_id":{"\_id":"5fe221b6573f5ff261265827"},"eno":300.0,"ename":"Chinny","esal":3000.0,"eaddr":"Chennai"}**

**{"\_id":{"\_id":"5fe221b6573f5ff261265828"},"eno":400.0,"ename":"Vinnny","esal":4000.0,"eaddr":"Delhi"}**

**{"\_id":{"\_id":"5fe222ac573f5ff261265829"},"eno":777.0,"ename":"Sunny","esal":1000.0,"eaddr":"Mumbai"}**

**{"\_id":{"\_id":"5fe222e9573f5ff26126582a"},"eno":888.0,"ename":"Bunny","esal":2000.0,"eaddr":"Hyd"}**



```
{"_id":{"_id":"5fe222e9573f5ff26126582b"},"eno":999.0,"ename":"Chin  
ny","esal":3000.0,"eaddr":"Chennai"}  
{"_id":{"_id":"5fe2241b573f5ff26126582c"},"eno":77777.0,"ename":"Sa  
chin","esal":99999.0,"eaddr":"Mumbai"}
```

**MongoDB backup and restore by using mongodump and mongorestore tools:**

---

**By using mongodump and mongorestore commands we can take backup of database and we can restore the backup data.**

**mongodump ==>Create a dump from mongodb database.**

**mongorestore ==>To restore data from dump.**

**Case-1: To create dump for all databases:**

---

**We have to use mongodump command without any arguments.**

**C:\Users\lenovo\Desktop>mongodump**

**2021-02-14T09:32:01.762+0530 writing admin.system.version to  
dump\admin\system.version.bson**

**2021-02-14T09:32:01.764+0530 done dumping admin.system.version  
(1 document)**

**2021-02-14T09:32:01.765+0530 writing durgadb.employees3 to  
dump\durgadb\employees3.bson**

**2021-02-14T09:32:01.773+0530 done dumping durgadb.employees3  
(1 document)**

**2021-02-14T09:32:01.773+0530 writing durgadb.employees2 to  
dump\durgadb\employees2.bson**

**2021-02-14T09:32:01.775+0530 done dumping durgadb.employees2  
(0 documents)**

**2021-02-14T09:32:01.776+0530 writing studentdb.sssdb1 to  
dump\studentdb\sssdb1.bson**

**2021-02-14T09:32:01.777+0530**    **done dumping studentdb.sssdb1 (0 documents)**  
**2021-02-14T09:32:02.067+0530**    **writing durgadb.employees to dump\durgadb\employees.bson**  
**2021-02-14T09:32:02.067+0530**    **writing studentdb.sssdb2 to dump\studentdb\sssdb2.bson**  
**2021-02-14T09:32:02.067+0530**    **writing studentdb.students to dump\studentdb\students.bson**  
**2021-02-14T09:32:02.070+0530**    **done dumping studentdb.students (1 document)**  
**2021-02-14T09:32:02.075+0530**    **done dumping studentdb.sssdb2 (1 document)**  
**2021-02-14T09:32:02.075+0530**    **done dumping durgadb.employees (8 documents)**

**Note: dump folder got created which contains data from all databases.**  
**For every collection 2 files will be created.**  
**bson file==>contains original data in bson format**  
**json file==>contains metadata related to that collection.**

**C:\Users\lenovo\Desktop>tree /f dump**

**Folder PATH listing**

**Volume serial number is 00000220 526A:E31B**

**C:\USERS\LENOVO\DESKTOP\DUMP**

```

|——admin
|   system.version.bson
|   system.version.metadata.json
|
|——durgadb
|   employees.bson
|   employees.metadata.json
|   employees2.bson
|   employees2.metadata.json

```

```

|   employees3.bson
|   employees3.metadata.json
|
└── studentdb
    sssdb1.bson
    sssdb1.metadata.json
    sssdb2.bson
    sssdb2.metadata.json
    students.bson
    students.metadata.json

```

**Note:**

The original data stored in bson form where as metadata stored in json form.

But we can convert bson data to json format by using bsondump tool.

**bsondump ==>To convert data from bson format to json format**

**eg-1: Convert bson data from employees.bson to json form and display to the console.**

```

C:\Users\lenovo\Desktop\dump\durgadb>bsondump employees.bson
{"_id":{"_id":{"$oid":"5fe220cd573f5ff261265825"},"eno":{"$numberDouble":"100.0"},"ename":"Sunny","esal":{"$numberDouble":"1000.0"},"eaddr":"Mumbai"}}
{"_id":{"_id":{"$oid":"5fe221b6573f5ff261265826"},"eno":{"$numberDouble":"200.0"},"ename":"Bunny","esal":{"$numberDouble":"2000.0"},"eaddr":"Hyd"}
{"_id":{"_id":{"$oid":"5fe221b6573f5ff261265827"},"eno":{"$numberDouble":"300.0"},"ename":"Chinny","esal":{"$numberDouble":"3000.0"},"eaddr":"Chennai"}
{"_id":{"_id":{"$oid":"5fe221b6573f5ff261265828"},"eno":{"$numberDouble":"400.0"},"ename":"Vinny","esal":{"$numberDouble":"4000.0"},"eaddr":"Delhi"}

```

```

{"_id":{"$oid":"5fe222ac573f5ff261265829"},"eno":{"$numberDouble":"777.0"},"ename":"Sunny","esal":{"$numberDouble":"1000.0"},"eaddr":"Mumbai"}
{"_id":{"$oid":"5fe222e9573f5ff26126582a"},"eno":{"$numberDouble":"888.0"},"ename":"Bunny","esal":{"$numberDouble":"2000.0"},"eaddr":"Hyd"}
{"_id":{"$oid":"5fe222e9573f5ff26126582b"},"eno":{"$numberDouble":"999.0"},"ename":"Chinny","esal":{"$numberDouble":"3000.0"},"eaddr":"Chennai"}
{"_id":{"$oid":"5fe2241b573f5ff26126582c"},"eno":{"$numberDouble":"7777.0"},"ename":"Sachin","esal":{"$numberDouble":"99999.0"},"eaddr":"Mumbai"}
2021-02-14T09:41:01.199+0530 8 objects found

```

**eg-2: Convert bson data from employees.bson to json form and write to emp.json file**

**By default bsondump tool outputs data to the standard output(console). We can write converted data to the files also. For this we have to use -outFile argument.**

```
bsondump --outFile=emp.json employees.bson
```

```
C:\Users\lenovo\Desktop\dump\durgadb>bsondump --outFile=emp.json employees.bson
2021-02-14T09:44:29.662+0530 8 objects found
```

**emp.json:**

```

-----
{"_id":{"$oid":"5fe220cd573f5ff261265825"},"eno":{"$numberDouble":"1000.0"},"ename":"Sunny","esal":{"$numberDouble":"1000.0"},"eaddr":"Mumbai"}

```

```

{"_id":{"$oid":"5fe221b6573f5ff261265826"},"eno":{"$numberDouble":"200.0"},"ename":"Bunny","esal":{"$numberDouble":"2000.0"},"eaddr":"Hyd"}
{"_id":{"$oid":"5fe221b6573f5ff261265827"},"eno":{"$numberDouble":"300.0"},"ename":"Chinny","esal":{"$numberDouble":"3000.0"},"eaddr":"Chennai"}
{"_id":{"$oid":"5fe221b6573f5ff261265828"},"eno":{"$numberDouble":"400.0"},"ename":"Vinny","esal":{"$numberDouble":"4000.0"},"eaddr":"Delhi"}
{"_id":{"$oid":"5fe222ac573f5ff261265829"},"eno":{"$numberDouble":"777.0"},"ename":"Sunny","esal":{"$numberDouble":"1000.0"},"eaddr":"Mumbai"}
{"_id":{"$oid":"5fe222e9573f5ff26126582a"},"eno":{"$numberDouble":"888.0"},"ename":"Bunny","esal":{"$numberDouble":"2000.0"},"eaddr":"Hyd"}
{"_id":{"$oid":"5fe222e9573f5ff26126582b"},"eno":{"$numberDouble":"999.0"},"ename":"Chinny","esal":{"$numberDouble":"3000.0"},"eaddr":"Chennai"}
{"_id":{"$oid":"5fe2241b573f5ff26126582c"},"eno":{"$numberDouble":"7777.0"},"ename":"Sachin","esal":{"$numberDouble":"99999.0"},"eaddr":"Mumbai"}

```

```

mongoimport
mongoexport
mongodump
    bson format
    bsondump

```

**Case-2: To restore all databases from dump by using mongorestore tool:**

-----  
**We have to use mongorestore command without any arguments.**

**C:\Users\lenovo\Desktop>mongorestore**

**2021-02-15T09:22:32.927+0530 using default 'dump' directory**  
**2021-02-15T09:22:32.928+0530 preparing collections to restore from**  
**2021-02-15T09:22:32.930+0530 reading metadata for**  
**durgadb.employees from dump\durgadb\employees.metadata.json**  
**2021-02-15T09:22:32.930+0530 reading metadata for**  
**studentdb.students from dump\studentdb\students.metadata.json**  
**2021-02-15T09:22:32.930+0530 reading metadata for**  
**durgadb.employees3 from dump\durgadb\employees3.metadata.json**  
**2021-02-15T09:22:33.232+0530 restoring studentdb.students from**  
**dump\studentdb\students.bson**  
**2021-02-15T09:22:33.232+0530 reading metadata for**  
**studentdb.sssdb2 from dump\studentdb\sssdb2.metadata.json**  
**2021-02-15T09:22:33.243+0530 no indexes to restore**  
**2021-02-15T09:22:33.243+0530 finished restoring studentdb.students**  
**(1 document, 0 failures)**  
**2021-02-15T09:22:33.243+0530 reading metadata for**  
**studentdb.sssdb1 from dump\studentdb\sssdb1.metadata.json**  
**2021-02-15T09:22:33.756+0530 restoring durgadb.employees3 from**  
**dump\durgadb\employees3.bson**  
**2021-02-15T09:22:33.767+0530 no indexes to restore**  
**2021-02-15T09:22:33.767+0530 finished restoring**  
**durgadb.employees3 (1 document, 0 failures)**  
**2021-02-15T09:22:33.767+0530 reading metadata for**  
**durgadb.employees2 from dump\durgadb\employees2.metadata.json**  
**2021-02-15T09:22:33.823+0530 restoring durgadb.employees from**  
**dump\durgadb\employees.bson**  
**2021-02-15T09:22:33.840+0530 no indexes to restore**  
**2021-02-15T09:22:33.840+0530 finished restoring**  
**durgadb.employees (8 documents, 0 failures)**

**2021-02-15T09:22:33.934+0530** restoring studentdb.sssdb2 from  
 dump\studentdb\ssssdb2.bson  
**2021-02-15T09:22:33.952+0530** no indexes to restore  
**2021-02-15T09:22:33.952+0530** finished restoring studentdb.sssdb2  
 (1 document, 0 failures)  
**2021-02-15T09:22:34.012+0530** restoring studentdb.sssdb1 from  
 dump\studentdb\ssssdb1.bson  
**2021-02-15T09:22:34.026+0530** no indexes to restore  
**2021-02-15T09:22:34.026+0530** finished restoring studentdb.sssdb1  
 (0 documents, 0 failures)  
**2021-02-15T09:22:34.176+0530** restoring durgadb.employees2 from  
 dump\durgadb\employees2.bson  
**2021-02-15T09:22:34.192+0530** no indexes to restore  
**2021-02-15T09:22:34.192+0530** finished restoring  
 durgadb.employees2 (0 documents, 0 failures)  
**2021-02-15T09:22:34.192+0530** 11 document(s) restored  
 successfully. 0 document(s) failed to restore.

**Q. sir is it possible to auto dump everyday if we didnt dump manually?**  
**cronjobs/schedulers ==>python script**

### **Case-3: How to backup and restore a single database?**

**-----**  
**We have to specify required database as argument to mongodump command.**

**mongodump --db durgadb**

**C:\Users\lenovo\Desktop>mongodump --db durgadb**

**2021-02-15T09:30:28.759+0530** writing durgadb.employees3 to  
 dump\durgadb\employees3.bson  
**2021-02-15T09:30:28.761+0530** done dumping durgadb.employees3  
 (1 document)

**2021-02-15T09:30:29.058+0530**    writing durgadb.employees2 to  
 dump\durgadb\employees2.bson  
**2021-02-15T09:30:29.060+0530**    writing durgadb.employees to  
 dump\durgadb\employees.bson  
**2021-02-15T09:30:29.061+0530**    done dumping durgadb.employees2  
 (0 documents)  
**2021-02-15T09:30:29.062+0530**    done dumping durgadb.employees (8  
 documents)

**C:\Users\lenovo\Desktop>tree /f dump**

**Folder PATH listing**

**Volume serial number is 000001A3 526A:E31B**

**C:\USERS\LENOVO\DESKTOP\DUMP**

```

└──durgadb
    employees.bson
    employees.metadata.json
    employees2.bson
    employees2.metadata.json
    employees3.bson
    employees3.metadata.json
  
```

**To restore a single database:**

-----

**mongorestore --db durgadb dump\durgadb**

**C:\Users\lenovo\Desktop>mongorestore --db durgadb dump\durgadb**

**2021-02-15T09:34:35.985+0530**    The --db and --collection flags are  
 deprecated for this use-case; please use --nsInclude instead, i.e. with -  
 -nsInclude=\${DATABASE}.\${COLLECTION}

**2021-02-15T09:34:35.986+0530**    building a list of collections to  
 restore from dump\durgadb dir



2021-02-15T09:34:35.988+0530 reading metadata for  
 durgadb.employees from dump\durgadb\employees.metadata.json  
 2021-02-15T09:34:35.989+0530 reading metadata for  
 durgadb.employees3 from dump\durgadb\employees3.metadata.json  
 2021-02-15T09:34:35.989+0530 reading metadata for  
 durgadb.employees2 from dump\durgadb\employees2.metadata.json  
 2021-02-15T09:34:36.172+0530 restoring durgadb.employees3 from  
 dump\durgadb\employees3.bson  
 2021-02-15T09:34:36.184+0530 no indexes to restore  
 2021-02-15T09:34:36.184+0530 finished restoring  
 durgadb.employees3 (1 document, 0 failures)  
 2021-02-15T09:34:36.556+0530 restoring durgadb.employees from  
 dump\durgadb\employees.bson  
 2021-02-15T09:34:36.568+0530 no indexes to restore  
 2021-02-15T09:34:36.568+0530 finished restoring  
 durgadb.employees (8 documents, 0 failures)  
 2021-02-15T09:34:36.661+0530 restoring durgadb.employees2 from  
 dump\durgadb\employees2.bson  
 2021-02-15T09:34:36.680+0530 no indexes to restore  
 2021-02-15T09:34:36.680+0530 finished restoring  
 durgadb.employees2 (0 documents, 0 failures)  
 2021-02-15T09:34:36.680+0530 9 document(s) restored successfully.  
 0 document(s) failed to restore.

#### **Case-4: How to backup and restore a single collection:**

-----

**mongodump --db durgadb --collection employees**

**C:\Users\lenovo\Desktop>mongodump --db durgadb --collection  
employees**

**2021-02-15T09:37:26.592+0530** writing durgadb.employees to  
dump\durgadb\employees.bson  
**2021-02-15T09:37:26.594+0530** done dumping durgadb.employees (8  
documents)

**C:\Users\lenovo\Desktop>tree /f dump**

**Folder PATH listing**

**Volume serial number is 000001B4 526A:E31B**

**C:\USERS\LENOVO\DESKTOP\DUMP**

├──durgadb  
│ employees.bson  
│ employees.metadata.json

**Restore a particular collection:**

-----

**mongorestore --db durgadb --collection employees**  
**dump\durgadb\employees.bson**

**C:\Users\lenovo\Desktop>mongorestore --db durgadb --collection**  
**employees dump\durgadb\employees.bson**

**2021-02-15T09:40:50.660+0530** checking for collection data in  
dump\durgadb\employees.bson

**2021-02-15T09:40:50.663+0530** reading metadata for  
durgadb.employees from dump\durgadb\employees.metadata.json

**2021-02-15T09:40:50.813+0530** restoring durgadb.employees from  
dump\durgadb\employees.bson

**2021-02-15T09:40:50.856+0530** no indexes to restore

**2021-02-15T09:40:50.856+0530** finished restoring  
durgadb.employees (8 documents, 0 failures)

**2021-02-15T09:40:50.857+0530** 8 document(s) restored successfully.  
0 document(s) failed to restore.

**Note: We can take dump and restore all databases or a particular database or a particular collection.**

**Summary:**

-----

- 1. mongoimport --->To import data from the files**
- 2. mongoexport --->To export mongodb data to the files**
- 3. mongodump --->To create dump for database**
- 4. bsondump --->To convert bson data into json data**
- 5. mongorestore --->To restore data from the dump.**

**Q. Sir, one last question: I want to know how many databases contain "books" collection. How can I perform a database level search from mongo shell?**

**python script:**

**show dbs**

**use every db**

**show collections**

**if books collection is there then print database name**

**GUI Tools for MongoDB Operations:**

-----

**Upto this we performed the database operations by using mongo shell. The advantage of using shell is we have to do everything so that learning opportunity is more.**

**But in real time usage of shell is not recommended because of the following reasons:**

- 1. Auto completion is not available. We have to type complete command.**
- 2. While writing complex queries, more error prone.**

- 3. No help tips**
- 4. Readability is not up to the mark.**
- 5. No coloring**
- 6. Operations will become very complex**
- 7. Not that much convenient to use**
- etc**

**To overcome these problems we have to use GUI based tools like**

- 1. Robo 3T**
- 2. Studio 3T**
- 3. Compass**
- 4. NoSQL Manager**
- etc**

#### **1. Robo 3T:**

-----

**The Robo 3T (formerly Robomongo) tool has been acquired by 3T Software Labs, the creators of the MongoDB client Studio 3T (formerly MongoChef).**

**It is freeware and lightweight GUI tool for MongoDB operations.**

**website: [robomongo.org](http://robomongo.org)**

**Two tools: Robo 3T and Studio 3T**

**Robo 3T is freeware and Studio 3T is licensed.**

**Download: [studio-3t-robo-3t-windows-double-pack.zip](#)**

**Create a new database**

**Create a new collection**

**CRUD Operations**

## **2. Studio 3T:**

-----

**Formerly MongoChef.**

**The most popular DB Tool for MongoDB operations.**

**<https://robomongo.org/>**

**studio-3t-x64.exe**

## **3. NoSQL Manager for MongoDB:**

-----

**NoSQL Manager for MongoDB Desktop tool for Mongo database management, administration and development**

**<https://www.mongodbmanager.com/>**

**NoSQL Manager for MongoDB Freeware**

**Free for non-commercial use, limited functional**

**mongodbmanagerfree\_inst.exe**

### **Features**

**Full support for MongoDB and MongoDB Enterprise versions from 2.2 to 4.4**

**Certified on MongoDB Enterprise. Supports Kerberos, LDAP and MONGODB-X509 authentications**

**Fully functional MongoDB UI Shell with code autocompletion, syntax highlighting and hints**

**Fully compatible with Amazon DocumentDB and MongoDB on Azure Cosmos DB**

**Support of replica sets, standalone hosts and sharded clusters connections**

**Easy-to-use document viewer and editor with Tree, Table and JSON view modes**

**Simple view and management of all MongoDB objects: databases, collections, views, indices, users, roles and functions**

**SSH tunneling for MongoDB connections**

**Performance monitoring tools**

**Import tables from MySQL and SQL Server databases**

**Export documents to CSV, XML, JSON and XLSX file formats**

**Import documents from JSON and CSV files**

**LINQ Query Tool**

#### **4. MongoDB Compass:**

-----

**<https://www.mongodb.com/products/compass>**

**The GUI for MongoDB.**

**The easiest way to explore and manipulate your MongoDB data.**

**Visually explore your data.**

**Run ad hoc queries in seconds.**

**Interact with your data with full CRUD functionality.**

**View and optimize your query performance.**

**Available on Linux, Mac, or Windows.**

**Compass empowers you to make smarter decisions about indexing, document validation, and more.**

**Connection String:**

**mongodb://localhost:27017**

**MongoDB Indexing:**

-----

**find()**  
**update({}, {})**  
**delete({})**

**We have to find matched documents.**

**book without index**  
**book with index**

**goto chapter11**

**shopping 100s of employees**

**If i need to run only one query from very large data, then do we need to create index as it will also takes time**

**Book my show**  
**select all movies in given theatre and in a particular city**  
**IRCTC**

**select all trains from source to destination**

**100000 trains**  
**Hyderabad**

```

db.employees.find({ename: "Sunny"}).pretty()
db.employees.find({ename: "Sunny"}).explain("executionStats")

> db.employees.find({ename: "Sunny"}).explain("executionStats")
{
  "queryPlanner" : {
    "plannerVersion" : 1,
    "namespace" : "durgadb.employees",
    "indexFilterSet" : false,
    "parsedQuery" : {
      "ename" : {
        "$eq" : "Sunny"
      }
    },
    "winningPlan" : {
      "stage" : "COLLSCAN",
      "filter" : {
        "ename" : {
          "$eq" : "Sunny"
        }
      },
      "direction" : "forward"
    },
    "rejectedPlans" : [ ]
  },
  "executionStats" : {
    "executionSuccess" : true,
    "nReturned" : 2,
    "executionTimeMillis" : 0,
    "totalKeysExamined" : 0,
    "totalDocsExamined" : 13,
    "executionStages" : {
      "stage" : "COLLSCAN",

```



```

        "filter" : {
            "ename" : {
                "$eq" : "Sunny"
            }
        },
        "nReturned" : 2,
        "executionTimeMillisEstimate" : 0,
        "works" : 15,
        "advanced" : 2,
        "needTime" : 12,
        "needYield" : 0,
        "saveState" : 0,
        "restoreState" : 0,
        "isEOF" : 1,
        "direction" : "forward",
        "docsExamined" : 13
    }
},
"serverInfo" : {
    "host" : "DESKTOP-ECE8V3R",
    "port" : 27017,
    "version" : "4.4.2",
    "gitVersion" :
"15e73dc5738d2278b688f8929aee605fe4279b0e"
},
"ok" : 1
}

```

**Define index and then find():**

-----

**db.employees.getIndexes()**  
**Returns available indexes**

```
> db.employees.getIndexes()
[ { "v" : 2, "key" : { "_id" : 1 }, "name" : "_id_" } ]
```

```
db.employees.createIndex({ename: 1})
```

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

```
> db.employees.getIndexes()
[
  {
    "v" : 2,
    "key" : {
      "_id" : 1
    },
    "name" : "_id_"
  },
  {
    "v" : 2,
    "key" : {
      "ename" : 1
    },
    "name" : "ename_1"
  }
]
```

```

> db.employees.find({ename: "Sunny"}).explain("executionStats")
{
  "queryPlanner" : {
    "plannerVersion" : 1,
    "namespace" : "durgadb.employees",
    "indexFilterSet" : false,
    "parsedQuery" : {
      "ename" : {
        "$eq" : "Sunny"
      }
    },
    "winningPlan" : {
      "stage" : "FETCH",
      "inputStage" : {
        "stage" : "IXSCAN",
        "keyPattern" : {
          "ename" : 1
        },
        "indexName" : "ename_1",
        "isMultiKey" : false,
        "multiKeyPaths" : {
          "ename" : [ ]
        },
        "isUnique" : false,
        "isSparse" : false,
        "isPartial" : false,
        "indexVersion" : 2,
        "direction" : "forward",
        "indexBounds" : {
          "ename" : [
            ["Sunny\","Sunny\"]
          ]
        }
      }
    }
  }
}

```

```

    }
  },
  "rejectedPlans" : [ ]
},
"executionStats" : {
  "executionSuccess" : true,
  "nReturned" : 2,
  "executionTimeMillis" : 0,
  "totalKeysExamined" : 2,
  "totalDocsExamined" : 2,
  "executionStages" : {
    "stage" : "FETCH",
    "nReturned" : 2,
    "executionTimeMillisEstimate" : 0,
    "works" : 3,
    "advanced" : 2,
    "needTime" : 0,
    "needYield" : 0,
    "saveState" : 0,
    "restoreState" : 0,
    "isEOF" : 1,
    "docsExamined" : 2,
    "alreadyHasObj" : 0,
    "inputStage" : {
      "stage" : "IXSCAN",
      "nReturned" : 2,
      "executionTimeMillisEstimate" : 0,
      "works" : 3,
      "advanced" : 2,
      "needTime" : 0,
      "needYield" : 0,
      "saveState" : 0,
      "restoreState" : 0,

```

```

        "isEOF" : 1,
        "keyPattern" : {
            "ename" : 1
        },
        "indexName" : "ename_1",
        "isMultiKey" : false,
        "multiKeyPaths" : {
            "ename" : [ ]
        },
        "isUnique" : false,
        "isSparse" : false,
        "isPartial" : false,
        "indexVersion" : 2,
        "direction" : "forward",
        "indexBounds" : {
            "ename" : [
                ["Sunny\\", \\Sunny\\"]
            ]
        },
        "keysExamined" : 2,
        "seeks" : 1,
        "dupsTested" : 0,
        "dupsDropped" : 0
    }
}
},
"serverInfo" : {
    "host" : "DESKTOP-ECE8V3R",
    "port" : 27017,
    "version" : "4.4.2",
    "gitVersion" :
"15e73dc5738d2278b688f8929aee605fe4279b0e"
},

```

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

**We can define index on multiple fields:**

```
db.employees.createIndex({eno:1,ename:-1})
```

**How to drop index?**

-----

```
db.employees.dropIndex({ename: 1})
```

```
> db.employees.dropIndex({ename: 1})
{ "nIndexesWas" : 2, "ok" : 1 }
> db.employees.getIndexes()
[ { "v" : 2, "key" : { "_id" : 1 }, "name" : "_id_" } ]
```

**Indexing is DBMS related concept and almost every database provides support.**

**Indexes supports efficient execution of queries in MongoDB.**

**Indexing concept is very helpful for find,update and delete queries.**

**The main objective of indexing is to improve performance.**

**Without indexes, server will scan all documents present in the collection to find matched documents. This is called Collection Scan(COLLSCAN).**

**Whenever we are defining index, MongoDB Server will store values of indexed field in B-Tree Data structure in specified sorting order.**

**To find matched documents, we are not required to scan all documents,server can identify matched documents directly based Index Scan(IXSCAN).**

**As the number of documents to be scan, is reduces and hence performance will be improved.**

**How to check available indexes?**

-----

**db.collection.getIndexes()**

**How to create Index:**

-----

**db.collection.createIndex({field: 1|-1})**

**1 means Ascending order**

**-1 means Descending order**

**How to drop index:**

-----

**db.collection.dropIndex({field:1|-1})**

**Is it possible to define index on multiple fields:**

-----

**db.collection.createIndex({field1: 1, field2: -1})**

**The order of fields is important.**

**Q. Is it recommended to define index for all fields?**

**No, performance will be degraded.**

**Server has to store every indexed field value separately.**

**Q. Is it recommended to define index for small collections?**

**No. Instead of improving performance , it will be reduced.**

**This is like keeping multiple cash counters in paan shop.**

**Q. How to see execution stats of query?**

**explain() method with executionStats argument**

**executionStats** – this mode includes all the information provided by the queryPlanner, plus the statistics. Statistics include details such as the number of documents examined and returned, the execution time in milliseconds, and so on.

```
db.collection.find({query}).explain("executionStats")
```

**Q. Is any default index for our collection?**

**For every collection default index is available, and it is based on `_id` field.**

```
db.employees.getIndexes()  
[ { "v" : 2, "key" : { "_id" : 1 }, "name" : "_id_" } ]
```

**Aggregation Framework:**

-----

```
db.employees.insertOne({eno:100,ename:"Sunny",esal:1000,eaddr:"Mumbai"})  
db.employees.insertOne({eno:200,ename:"Bunny",esal:2000,eaddr:"Hyderabad"})  
db.employees.insertOne({eno:300,ename:"Chinny",esal:3000,eaddr:"Hyderabad"})  
db.employees.insertOne({eno:400,ename:"Vinny",esal:4000,eaddr:"Mumbai"})  
db.employees.insertOne({eno:500,ename:"Pinny",esal:5000,eaddr:"Chennai"})  
db.employees.insertOne({eno:600,ename:"Zinny",esal:6000,eaddr:"Chennai"})  
db.employees.insertOne({eno:700,ename:"Yinny",esal:7000,eaddr:"Hyderabad"})
```



```
db.employees.find().pretty()  
db.employees.find({eaddr:"Hyderabad"}).pretty()
```

**Total salary of all employees irrespective of city**

**Total salary of all employees city wise**

**Average salary of all employees city wise**

**minimum salary of employees city wise**

**maximum salary of employees city wise**

**What is the min salary of all employees**

**What is the max salary of all employees**

**etc**

**Such type of requirements won't be fulfilled by find() methods.**

**find() methods will provide existing data as it is. If we want to process documents and to provide results in our customized format then we should go for aggregation framework.**

**We can implement aggregation by using aggregate() method.**

**aggregate() method is more powerful than find() method.**

**aggregate() vs find():**

**-----**

**aggregate() method can perform some processing and provide results in our customized format.**

**But find() method will always provide data as it is without performing any processing and in the existing format only.**

**eg-1: To find total salary of all employees?**

```
db.employees.aggregate([
```

```
{ $group: { _id:null,totalsalary:{$sum:"$esal"}}}  
}
```

```
{ "_id" : null, "totalsalary" : 28000 }
```

**Note:**

**\$group stage:**

-----

- 1. It is the most important stage.**
- 2. It can be used to group documents based on required fields. It merges different documents into new documents.**  
**eg: group all documents city wise**  
**group all documents department wise**
- 3. The first parameter in \$group stage is always \_id.**
- 4. We should use \_id to specify field based on which we have to perform grouping. ie \_id defines by which fields we can group.**
- 5. If we want to process all records then we have to provide null value to \_id field.**
- 6. \$group is exactly same as group by phrase in relational databases.**

**Accumulator operators:**

-----

**These operators can be used for accumulation purpose.**

**The following various accumulator operators:**

- 1. \$sum: Returns a sum of numerical values. Ignores non-numeric values.**
  - 2. \$avg: Returns an average of numerical values. Ignores non-numeric values.**
  - 3. \$max: Returns the highest expression value for each group.**
  - 4. \$min: Returns the lowest expression value for each group.**
- etc**

**eg-2: To find average salary of all employees?**

```
db.employees.aggregate([  
  { $group: { _id:null,averagesalary:{$avg:"$esal"}}}  
])
```

```
{ "_id" : null, "averagesalary" : 4000 }
```

**eg-3: To find max salary of all employees?**

```
db.employees.aggregate([  
  { $group: { _id:null,maxsalary:{$max:"$esal"}}}  
])
```

```
{ "_id" : null, "maxsalary" : 7000 }
```

**eg-4: To find min salary of all employees?**

```
db.employees.aggregate([  
  { $group: { _id:null,minsalar:{$min:"$esal"}}}  
])
```

```
{ "_id" : null, "minsalar" : 1000 }
```

**eg-5: To find max salary city wise?**

**We have to group documents city wise. ie based on eaddr field.**

```
db.employees.aggregate([  
  {$group: { _id:"$eaddr",maxSalary:{$max:"$esal"}}}  
])
```

**o/p:**

```
{ "_id" : "Mumbai", "maxSalary" : 4000 }  
{ "_id" : "Hyderabad", "maxSalary" : 7000 }  
{ "_id" : "Chennai", "maxSalary" : 6000 }
```

**eg-6: To find city wise total salary?**

```
db.employees.aggregate([  
  {$group: {_id:"$eaddr",totalSalary:{$sum:"$esal"}}}  
])
```

**o/p:**

```
{ "_id" : "Mumbai", "totalSalary" : 5000 }  
{ "_id" : "Hyderabad", "totalSalary" : 12000 }  
{ "_id" : "Chennai", "totalSalary" : 11000 }
```

**eg-7: To find city wise average salary?**

```
db.employees.aggregate([  
  {$group: {_id:"$eaddr",averageSalary:{$avg:"$esal"}}}  
])
```

**o/p:**

```
{ "_id" : "Mumbai", "averageSalary" : 2500 }  
{ "_id" : "Hyderabad", "averageSalary" : 4000 }  
{ "_id" : "Chennai", "averageSalary" : 5500 }
```

**eg-8: To find total number of employees?**

```
db.employees.aggregate([  
  {$group: {_id:null,employeecount:{$sum:1}}}  
])
```

**For every document add 1 to the employeecount.**

**o/p: { "\_id" : null, "employeecount" : 7 }**

**eg-9: To find total number of employees city wise?**

```
db.employees.aggregate([  
  {$group: {_id:"$eaddr",employeecount:{$sum:1}}}  
])
```

**o/p:**

```
{ "_id" : "Mumbai", "employeecount" : 2 }  
{ "_id" : "Hyderabad", "employeecount" : 3 }  
{ "_id" : "Chennai", "employeecount" : 2 }
```

### **Aggregation Pipeline:**

-----

**We can define multiple stages in the aggregation and all these stages will form pipeline, which is known as aggregation pipeline.**

```
db.collection.aggregate([  
  {stage-1},  
  {stage-2},  
  {stage-3},  
  {stage-4},  
  {stage-5}  
  ...  
)
```

**All these stages will be executed one by one.**

**The output of previous stage will become input to next stage.**

**Pipeline will take collection as input and provides aggregated results in our required format.**

**Note: In Documentation just explore as many stages as possible.**

**reference-->Operators--->Aggregation Pipeline stages**

### **2. \$sort stage:**

-----

**It sorts all input documents and returns them to the pipeline in sorted order.**

**The \$sort stage has the following prototype form:**

**{ \$sort: { <field1>: <sort order>, <field2>: <sort order> ... } }**

**The <sort order> can be either 1 or -1.**

**1 --->Ascending Order**

**-1 ---> Descending Order**

**eg-1: Find citywise sum of salaries and print based on descending order of totalsalary?**

```
db.employees.aggregate([  
  { $group: { _id:"$eaddr",totalSalary:{$sum:"$esal"} }},  
  { $sort:{totalSalary: -1}}  
])
```

**o/p:**

```
{ "_id" : "Hyderabad", "totalSalary" : 12000 }  
{ "_id" : "Chennai", "totalSalary" : 11000 }  
{ "_id" : "Mumbai", "totalSalary" : 5000 }
```

**eg-2: Find citywise number of employees and print based on alphabetical order of city name?**

```
db.employees.aggregate([  
  { $group: { _id:"$eaddr", employeeCount: {$sum:1}}},  
  { $sort: { _id:1}}  
])
```

**o/p:**

```
{ "_id" : "Chennai", "employeeCount" : 2 }  
{ "_id" : "Hyderabad", "employeeCount" : 3 }  
{ "_id" : "Mumbai", "employeeCount" : 2 }
```

### **3. \$project stage:**

-----

**By using this \$project stage, we can restrict documents with our required fields only. Here we can include new fields also. With the existing field values we can create new fields also.**

**eg: first name and last name fields are there. We can combined these fields with just name field.**

#### **Syntax:**

```
{ $project: { field:0|1 } }
```

**0 or false --->To exclude the field**

**1 or true --->To include the field**

**eg-1: To find total salary of all employees?**

```
db.employees.aggregate([  
  { $group: { _id:null, totalSalary:{$sum:"$esal"}},  
  { $project: { _id:0}}  
])
```

**o/p:**

```
{ "totalSalary" : 28000 }
```

**eg-2: Find city wise total salary and city name should be in uppercase and sort the documents in ascending order of salaries?**

```
db.employees.aggregate([  
  {$group: { _id:"$eaddr",totalSalary:{$sum:"$esal"}},
```

```
{ $project: { _id:0,city:{$toUpper:"$_id"},totalSalary:1}},  
  { $sort: {totalSalary: 1}}  
}
```

**o/p:**

```
{ "totalSalary" : 5000, "city" : "MUMBAI" }  
{ "totalSalary" : 11000, "city" : "CHENNAI" }  
{ "totalSalary" : 12000, "city" : "HYDERABAD" }
```

```
db.employees.aggregate([  
  { $group: { _id:"$eaddr",totalSalary:{$sum:"$esal"} }},  
  { $project: { _id:0,city:{$concat:["$_id"," ","City"]},totalSalary:1}},  
  { $sort: {totalSalary: 1}}  
])
```

**o/p:**

```
{ "totalSalary" : 5000, "city" : "Mumbai City" }  
{ "totalSalary" : 11000, "city" : "Chennai City" }  
{ "totalSalary" : 12000, "city" : "Hyderabad City" }
```

**\$group vs \$project:**

-----

**If we want to group multiple documents into a single document then we should use \$group stage.**

**eg: \$sum,\$avg,\$max etc**

**But if we want to include or exclude existing fields and to create new fields with new values or already existing field values then we can use \$project. Here the number of input documents and output documents are always same. But number of fields may be changed.**

**4. \$match stage:**

-----



**To filter documents based on required condition. It is exactly same as find() method <query>.**

**Syntax:**

**{ \$match: { <query> } }**

**eg: To find the number of employees whose salary greater than 1500. Find such employees count citywise. Display documents in ascending order of employee count.**

```
db.employees.aggregate([  
  {$match: {esal:{$gt: 1500}}},  
  {$group: {_id:"$eaddr",employeeCount:{$sum:1}}},  
  {$sort: {employeeCount:1}}  
])
```

**o/p:**

```
{ "_id" : "Mumbai", "employeeCount" : 1 }  
{ "_id" : "Chennai", "employeeCount" : 2 }  
{ "_id" : "Hyderabad", "employeeCount" : 3 }
```