Mongo DB Essentials
Lab Book

MongoDB Exercise in mongo shell

Connect to a running mongo instance, use a database named `mongo_practice`.

Document all your queries in a javascript file to use as a reference.

Insert Documents
Insert the following documents into a `movies` collection.

```
title : Fight Club
writer : Chuck Palahniuk
year : 1999
actors : [
  Brad Pitt
  Edward Norton
]
title : Pulp Fiction
writer : Quentin Tarantino
year : 1994
actors : [
  John Travolta
  Uma Thurman
]
title : Inglorious Basterds
writer : Quentin Tarantino
year : 2009
actors : [
  Brad Pitt
  Diane Kruger
  Eli Roth
]
title : The Hobbit: An Unexpected Journey
writer : J.R.R. Tolkein
year : 2012
franchise : The Hobbit
title : The Hobbit: The Desolation of Smaug
writer : J.R.R. Tolkein
year : 2013
franchise : The Hobbit
title : The Hobbit: The Battle of the Five Armies
```

```
writer : J.R.R. Tolkein
year : 2012
franchise : The Hobbit
synopsis : Bilbo and Company are forced to engage in a war against an
array of combatants and keep the Lonely Mountain from falling into the
hands of a rising darkness.
title : Pee Wee Herman's Big Adventure
title : Avatar
```

Query / Find Documents

query the `movies` collection to

1. get all documents
2. get all documents with `writer` set to "Quentin Tarantino"
3. get all documents where `actors` include "Brad Pitt"
4. get all documents with `franchise` set to "The Hobbit"
5. get all movies released in the 90s
6. get all movies released before the year 2000 or after 2010

Update Documents

1. add a synopsis to "The Hobbit: An Unexpected Journey" : "A reluctant hobbit, Bilbo Baggins, sets out to the Lonely Mountain with a spirited group of dwarves to reclaim their mountain home - and the gold within it - from the dragon Smaug."
2. add a synopsis to "The Hobbit: The Desolation of Smaug" : "The dwarves, along with Bilbo Baggins and Gandalf the Grey, continue their quest to reclaim Erebor, their homeland, from Smaug. Bilbo Baggins is in possession of a mysterious and magical ring."
3. add an actor named "Samuel L. Jackson" to the movie "Pulp Fiction"

Text Search

1. find all movies that have a synopsis that contains the word "Bilbo"
2. find all movies that have a synopsis that contains the word "Gandalf"
3. find all movies that have a synopsis that contains the word "Bilbo" and not the word "Gandalf"
4. find all movies that have a synopsis that contains the word "dwarves" or "hobbit"
5. find all movies that have a synopsis that contains the word "gold" and "dragon"

Delete Documents

1. delete the movie "Pee Wee Herman's Big Adventure"
2. delete the movie "Avatar"

Relationships

Insert the following documents into a `users` collection

```
username : GoodGuyGreg
first_name : "Good Guy"
last_name : "Greg"
username : ScumbagSteve
full_name :
  first : "Scumbag"
  last : "Steve"
```

Insert the following documents into a `posts` collection

```
username : GoodGuyGreg
title : Passes out at party
body : Wakes up early and cleans house
username : GoodGuyGreg
title : Steals your identity
body : Raises your credit score
username : GoodGuyGreg
title : Reports a bug in your code
body : Sends you a Pull Request
username : ScumbagSteve
title : Borrows something
body : Sells it
username : ScumbagSteve
title : Borrows everything
body : The end
username : ScumbagSteve
title : Forks your repo on github
body : Sets to private
```

Insert the following documents into a `comments` collection

```
username : GoodGuyGreg
comment : Hope you got a good deal!
post : [post_obj_id]
```

where [post_obj_id] is the ObjectId of the `posts` document: "Borrows something"

```
username : GoodGuyGreg
comment : What's mine is yours!
post : [post_obj_id]
```

where [post_obj_id] is the ObjectId of the `posts` document: "Borrows everything"

```
username : GoodGuyGreg
comment : Don't violate the licensing agreement!
post : [post_obj_id]
```

where [post_obj_id] is the ObjectId of the `posts` document: "Forks your repo on github

```
username : ScumbagSteve
comment : It still isn't clean
post : [post_obj_id]
```

where [post_obj_id] is the ObjectId of the `posts` document: "Passes out at party"

```
username : ScumbagSteve
comment : Denied your PR cause I found a hack
post : [post_obj_id]
```

where [post_obj_id] is the ObjectId of the `posts` document: "Reports a bug in your code"

Querying related collections

1. find all users
2. find all postsG
3. find all posts that was authored by "Good uyGreg"
4. find all posts that was authored by "ScumbagSteve"
5. find all comments
6. find all comments that was authored by "GoodGuyGreg"
7. find all comments that was authored by "ScumbagSteve"
8. find all comments belonging to the post "Reports a bug in your code"

Exercises on Crud Operations

1. Find scores not greater than 78

   db.scores.find({score:{$not : {$gt:78}}},{student:true,score:true})

2. Find the scores greater than 79 and less than 92 with the type essay .

db.scores.find({score:{$gte :79,$lte : 92},type:"essay"})

3. Find all documents with type: essay and score: 50 and only retrieve the *student* field?

3. To find the documents that have the profession fied

   Db..people.find({profession :{$exists:true}})

3. Display the document where the profession field does not exist

   db.people.find({profession : { $exists:false }})

4. Find all documents where the name field is a string

   db.people.find({name:{$type :2}})

5. To find the documents with the name field that has the letter 'a'

db.people.find({name:{$regex : "a"}})

6. **Find the documents with name ending in e**

   Db.people.find({name:{$regex : "e$"}})

7. **Find all those people who name ends with an e, or who had an age we can do so using $ operator**

   db.people.find({$or :[{ name: {$regex :"e$"}},{age : {$exists : true}}]})

8. **Find all documents in the *scores* collection where the *score* is less than 50 or greater than 90?**

   db.scores.find({$or :[{score : { $lt:50}} ,{ score:{ $gt :90 }}] })

9. **Find the documents with the name greater than c and that contains the letter 'a' in it**

   db.people.find({$and :[{name :{ $gt :"C" }},{name : {$regex :"a" }}]})

<u>Documents can nest</u>

Lets see an example

Db.users.insert ({ name : "richard",email :{ work :" richard@10gen.com", personal :"kreuter@example.com }})

Here email field is a nested document

```
> db.users.findOne()

{

        "_id" : ObjectId("56cc08c875913d0a9c0b0614"),

        "name" : "Richard Keauter",

        "city of birth" : "chicago",

        "favorite_color" : "red"

}
```

```
> db.users.find({email :{work :"richard@10gen.com",personal :
"kreuter@gmail.com"}});
```

```
{ "_id" : ObjectId("56d82de70ef02d0aec0aacb5"), "name" : "richard", "email" : {
"work" : "richard@10gen.com", "personal" : "kreuter@gmail.com" } }
```

```
db.users.find({email :{personal : "kreuter@gmail.com", work :
"richard@10gen.com"}});
```

```
db.users.find({email : {work :"richard@10gen.com"}})
```

The above query does not give any result becos the query will match only a document whose email field is exactly work:"Richard@10gen.com

To resolve the above issue , Mongodb has special syntax that allows to reach the inside of the email field for a specific embedded field regardless what are fields are present around . We call that as dot notation .

```
> db.users.find({"email.work" :"richard@10gen.com"})
```

```
{ "_id" : ObjectId("56d82de70ef02d0aec0aacb5"), "name" : "richard", "email" : {
"work" : "richard@10gen.com", "personal" : "kreuter@gmail.com" } }
```

The purpose of dot notation is to reach the inside of nested documents looking for a specific embedded piece of information

Bulk mode of Insert

```
> db.foo.bulk.insert([{"_id" : 0}, {"_id" : 1}, {"_id" : 2}])

BulkWriteResult({

        "writeErrors" : [ ],

        "writeConcernErrors" : [ ],

        "nInserted" : 3,

        "nUpserted" : 0,

        "nMatched" : 0,

        "nModified" : 0,

        "nRemoved" : 0,

        "upserted" : [ ]

})

> db.foo.find()

{ "_id" : 1, "price" : 1.99 }
{ "_id" : 2 }

{ "_id" : 2.9 }

{ "_id" : "hello" }

{ "_id" : ISODate("2016-04-29T06:24:45.625Z") }

{ "_id" : { "a" : "x", "b" : 2 } }

{ "_id" : ObjectId("5722fef262daae10e560d1a2"), "name" : "Bob" }

{ "_id" : ObjectId("57371b949bbb29d3e0ca8229"), "a" : 1, "b" : 2 }
```

Cursor

==========

```
cur=db.scores.find();null;

null
```

cur is a variable that holds on the cursor .cursor objects have variety of methods .

cur.hasNext() returns true as long as there is any document .

for example the HasNext method returns true so long as theres another document to visit on the cursor .

```
> cur=db.scores.find();null;

null
```

```
> cur.size()

3000

> cur.hasNext()

true

> cur.forEach(function(d){print(d.student)})

> cur.next()


while (cur.hasNext())printjson(cur.next())


> cur=db.scores.find();null;

null

> cur.limit(5)   --- prints the 5 docs in the cursor
```

Sort Operations

```
> db.scores.find({},{student:true}).sort({student:1})

> db.scores.find({},{student:true}).sort({student:-1})

> db.scores.find({},{_id:true}).sort({student:1})

> db.scores.find({},{student:true}).sort({student:-1})
> db.scores.find({},{score:true,student:true}).sort({student:1})
```

Limit and skip

```
> db.scores.find({},{student:true}).sort({student:1}).limit(3)

> db.scores.find({},{student:true}).sort({student:1}).skip(2).limit(3)
```

With Cursors

===========

```
> cur=db.scores.find();null;

null

> cur.sort({score:-1}).limit(3);null;

null

> while (cur.hasNext())printjson(cur.next());

> cur=db.scores.find();null;

null

> cur.sort({student:-1}).limit(10).skip(2);null;

null

> cur.size()

10

> while (cur.hasNext())printjson(cur.next());
> cur=db.scores.find();null;

null

> cur.size()

3000

> cur.sort({student:-1}).limit(3).skip(4);null;

null

> cur.size()

3

> while (cur.hasNext())printjson(cur.next());
```

Count Method

Find the number of documents that have the type “essay”

```
db.scores.count({type :"exam"})
```

Update Method:

```
db.people.find()
```

```
{ "_id" : ObjectId("562887a16000ded4b84f279f"), "age" : 30, "name" : "Smith",
"profession" : "hacker", "title" : "Dr" }

{ "_id" : ObjectId("5628b51c6000ded4b84f335b"), "name" : "Bob", "title" : "Dr"
}

{ "_id" : ObjectId("5628b5236000ded4b84f335c"), "name" : "Charlie", "title" :
"Dr" }

{ "_id" : ObjectId("5628b52b6000ded4b84f335d"), "name" : "Dave", "title" :
"Dr" }

{ "_id" : ObjectId("5628b5346000ded4b84f335e"), "name" : "Edgar", "title" :
"Dr" }

{ "_id" : ObjectId("5628b53b6000ded4b84f335f"), "name" : "Fred", "title" : "Dr"
}

{ "_id" : ObjectId("5628b8ae6000ded4b84f3360"), "name" : 43, "title" : "Dr" }

{ "_id" : ObjectId("5631ed9cfdad1bc4cff7b582"), "age" : 40, "name" : "George",
"title" : "Dr" }

{ "_id" : ObjectId("5628b5146000ded4b84f335a"), "age" : 61, "name" :
"William", "title" : "Dr" }

{ "ACAD" : "MD", "_id" : ObjectId("5628892f6000ded4b84f27a0"), "age" : 35,
"name" : "jones", "title" : "Dr" }

{ "_id" : "Smith", "age" : 30, "title" : "Dr" }

{ "_id" : "James", "age" : 30, "title" : "Dr" }

{ "_id" : ObjectId("5633390706a449cca998fd9b"), "name" : "Thompson", "title"
: "Dr" }
```

```
{ "_id" : ObjectId("56d67e7671aa3d5811a0bb95"), "name" : "Smith", "age" : 30,
"profession" : "hacker" }

{ "_id" : ObjectId("56d6850371aa3d5811a0bb96"), "name" : "Jones", "age" : 35,
"Profession" : "baker" }

{ "_id" : ObjectId("56d6d9b071aa3d5811a0c74f"), "name" : "Alice" }

{ "_id" : ObjectId("56d6d9dd71aa3d5811a0c750"), "name" : "charlie" }

{ "_id" : ObjectId("56d6d9e571aa3d5811a0c751"), "name" : "Bob" }

{ "_id" : ObjectId("56d6d9f271aa3d5811a0c752"), "name" : "Dave" }

{ "_id" : ObjectId("56d6d9fa71aa3d5811a0c753"), "name" : "Edgar" }

Type "it" for more
```

Modify the name and salary of the person

===

```
> db.people.update({name :"Smith" },{name : "Thompson",salary : 50000})
```

```
db.people.find()
```

Using the $set command

```
db.people.find()

db.people.update({name :”Alice”},{name :”Alice”,age :30 })
```

To update the age field for the name alice ,e need to kno other fields .. This kind of update is not really useful . Instead we can go for $set operator and update the corresponding field which we want to update ..

Ex .Find the document whose name field is Alice and set the age to 30 . If there is no age field , one such will be created .

```
> db.people.update({name:"Alice"},{$set : { age: 30 } } )

> db.people.find({name:"Alice"})

> db.people.update({name:"Alice"},{$set : { age: 31 } } )

> db.people.find({name:"Alice"})
```

Another method called $inc → increment is present .

$set will add or modify a field in a document .

$inc will modify a filed in a document

```
db.people.update({name:"Alice" },{ $inc : { age : 1 } } )

> db.people.find({name:"Alice"})

> db.people.update({name:"Bob"},{$inc : { age: 1 } } )

> db.people.find({name:"Bob"})
```

$unset operator

To remove a particular field from the document , that is to remove Jone's profession we do in using the below method.

```
> db.people.update({ name: "Jones"},{$unset : { Profession : 1 } } )

> db.people.find({name:"Jones"})
```

This operator is useful for

Schema change manipulations

The application requirement changes

To model certain kinds of transformations of data.

Instead of adding or updating a null value , we can remove the field using $unset operator

Rename Operator -->changes the name of the field

=================

db.a.save({_id:1,naem:"bob"})

db.a.update({_id:1},{$rename:{'naem':'Name'}})

Array based operations

=======================

db.a.update({_id:1},{$push:{things:'one'}})

db.a.find()

{"_id":1,"things":["one"]}

db.a.update({_id:1},{$push:{things:'two'}})

db.a.update({_id:1},{$push:{things:'three'}})

db.a.find()

{"_id":1,"things":["one","two","three"]}

db.a.update({_id:1},{$push:{things:'three'}})

db.a.find()

{"_id":1,"things":["one","two","three","three"]}

If we do not want three to be duplicated , we use

$addToset operator

==================

db.a.update({_id:1},{$addToSet:{things:'four'}})

db.a.find()

{"_id":1,"things":["one","two","three","three","four"]}

db.a.update({_id:1},{$addToSet:{things:'four'}})

db.a.find()

{"_id":1,"things":["one","two","three","three","four"]} (it did not add any element)

$pull

======

If we have to remove an element from the array ,we can use $pull operator

```
db.a.update({_id:1},{$pull:{things:'three'}})
db.a.find()
{"_id":1,"things":["one","two","four"]}
```

```
db.a.update({_id:1},{$addToSet:{things:'three'}})
```

```
db.a.find()
{"_id":1,"things":["one","two","four","three"]}
```

If we are not sure of the elements to pull out .. lets say we want to pull outthe last element from the array , we use pop operator .

```
db.a.find()
{"_id":1,"things":["one","two","four","three"]}
```

```
db.a.update({_id:1},{$pop:{things:1}})
db.a.find()

{"_id":1,"things":["one","two","four"}


db.a.update({_id:1},{$pop:{things:-1}}) ( to remove the first element )

db.a.find()

{"_id":1,"things":["two","four"}
```

all these operators work on arrays and not on strings(non arrays)

Multiupdate :

==============

```
db.a.find()

{"_id":1,"things":[1,2,3]}

{"_id":1,"things":[2,3]}

{"_id":1,"things":[3]}

{"_id":1,"things":[1,3]}


db.a.update({},{$push:{things:4}})
```

|

```
db.a.find()
{"_id":1,"things":[1,2,3,4]}
{"_id":1,"things":[2,3]}
{"_id":1,"things":[3]}
{"_id":1,"things":[1,3]}
```

Only one document is affected

if multiple documents have to be affected then we give the option multi

```
db.a.update({},{$push:{things:4}},{multi:true})
db.a.find()
{"_id":1,"things":[1,2,3,4,4]}
{"_id":1,"things":[2,3,4]}
{"_id":1,"things":[3,4]}
{"_id":1,"things":[1,3,4]} ( all updated)
```

If we have to update the elements for things array which has element 2 then

```
db.a.update({things:2},{$push:{things:42}},{multi:true})
```

db.a.find()

{"_id":3,"things":[3,4]}

{"_id":4,"things":[1,3,4]}

{"_id":1,"things":[1,2,3,4,4,42]} ---> updated

{"_id":2,"things":[1,3,4,42]} ----> updated

Find and modify

Upserts

In Mongo Shell the update operator does four different things :

Whole Cell replacement document

Manipulation of fields inside a document

The third one is upsert

It can update multiple documents

```
> db.people.update({name : "srilekha" },{$set : { age : 30 }},{upsert : true} )

WriteResult({

        "nMatched" : 0,

        "nUpserted" : 1,

        "nModified" : 0,

        "_id" : ObjectId("573ab9f18cc7c61fb3defdf2")

})

> db.people.find()

{ "_id" : ObjectId("573999bcb8bedb60cb42ff3f"), "name" : "akash" }

{ "_id" : 1, "name" : "Hemanth" }

{ "_id" : 2, "name" : "mani" }

{ "_id" : 3, "name" : "vish", "depno" : 30 }

{ "_id" : 5, "values" : [ 1, 2, 3 ] }

{ "_id" : ObjectId("573ab9f18cc7c61fb3defdf2"), "name" : "srilekha", "age" : 30
}

> db.people.update({name : "george" },{$set : { age : 30 }} )

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

> db.people.find()

{ "_id" : ObjectId("573999bcb8bedb60cb42ff3f"), "name" : "akash" }

{ "_id" : 1, "name" : "Hemanth" }
{ "_id" : 2, "name" : "mani" }

{ "_id" : 3, "name" : "vish", "depno" : 30 }

{ "_id" : 5, "values" : [ 1, 2, 3 ] }

{ "_id" : ObjectId("573ab9f18cc7c61fb3defdf2"), "name" : "srilekha", "age" : 30
}
```

```
db.people.update ({ age : {$gt : 50 }},{$set : { name : "srisuman" }},{upsert : true
} )
```

The above document gets created if not already existing

Upset is used rarely . For example if you are merging data in from a data vendor and you don't if the record exists and so we straight away use the upsert operator to either update the existing document or create a new document .

To update multiple documents

=======================

```
> db.people.update ({},{ $set : {title :"Dr"}},{multi : true } )

WriteResult({ "nMatched" : 7, "nUpserted" : 0, "nModified" : 7 })

> db.people.find()

{ "_id" : ObjectId("573999bcb8bedb60cb42ff3f"), "name" : "akash", "title" :
"Dr" }

{ "_id" : 1, "name" : "Hemanth", "title" : "Dr" }

{ "_id" : 2, "name" : "mani", "title" : "Dr" }

{ "_id" : 3, "name" : "vish", "depno" : 30, "title" : "Dr" }
{ "_id" : 5, "values" : [ 1, 2, 3 ], "title" : "Dr" }

{ "_id" : ObjectId("573ab9f18cc7c61fb3defdf2"), "name" : "srilekha", "age" : 30,
"title" : "Dr" }

{ "_id" : ObjectId("573abaae8cc7c61fb3defdf3"), "name" : "srisuman", "title" :
"Dr" }

> db.people.update ({},{ $set : {title :"Sr"}},{multi : true } )

> db.people.find()

{ "_id" : ObjectId("573999bcb8bedb60cb42ff3f"), "name" : "akash", "title" : "Sr"
}

{ "_id" : 1, "name" : "Hemanth", "title" : "Sr" }

{ "_id" : 2, "name" : "mani", "title" : "Sr" }

{ "_id" : 3, "name" : "vish", "depno" : 30, "title" : "Sr" }

{ "_id" : 5, "values" : [ 1, 2, 3 ], "title" : "Sr" }

{ "_id" : ObjectId("573ab9f18cc7c61fb3defdf2"), "name" : "srilekha", "age" : 30,
"title" : "Sr" }

{ "_id" : ObjectId("573abaae8cc7c61fb3defdf3"), "name" : "srisuman", "title" :
"Sr" }
```

Remove :

Remove is a method in a collection to remove a document . It works like find which takes the argument to specify what documents to remove . .

We must pass a document to remove . If we specify an empty document then it will remove all documents in a collection one by one . If we specify one argument as given below then only that document gets removed .

> db.people.remove ({ name : "Alice" })

db.people.remove({ name : {$gt : "M"} })

> db.people.find()

> db.people.remove() – no argument will remove all documents from the collection

> db.people.find()

Delete every document with a score of less than 60 .

db.scores.remove({ score : { $lt : 60 } })

Find and modify

Create an array and an object

===============================

> db.b.insert({"_id":1,"elements":[1,2,3]})

WriteResult({ "nInserted" : 1 })

> db.b.insert({"_id":2,"elements":[2,3]})

WriteResult({ "nInserted" : 1 })

> db.b.insert({"_id":3,"elements":[1,3]})

WriteResult({ "nInserted" : 1 })

```
> db.b.insert({"_id":4,"elements":[3]})

WriteResult({ "nInserted" : 1 })

> db.b.find()

{ "_id" : 1, "elements" : [ 1, 2, 3 ] }

{ "_id" : 2, "elements" : [ 2, 3 ] }

{ "_id" : 3, "elements" : [ 1, 3 ] }

{ "_id" : 4, "elements" : [ 3 ] }
```

- > mod =({"query":{"elements":1},"update":{"$set":{"touched":true}},"sort":{"_id":-1}})
- {
-         "query" : {
-                 "elements" : 1
-         },
-         "update" : {
-                 "$set" : {
-                         "touched" : true
-                 }
-         },
-         "sort" : {
-                 "_id" : -1
-         }
- }
- > db.b.findAndModify(mod)
- { "_id" : 3, "elements" : [1, 3] }

```
> db.b.find()
```

```
{ "_id" : 1, "elements" : [ 1, 2, 3 ] }

{ "_id" : 2, "elements" : [ 2, 3 ] }

{ "_id" : 3, "elements" : [ 1, 3 ], "touched" : true }

{ "_id" : 4, "elements" : [ 3 ] }
```

Returns the documents after it is modified

```
> mod.new=true

true

> mod

{

        "query" : {

                "elements" : 1

        },

        "update" : {

                "$set" : {

                        "touched" : true

                }

        },

        "sort" : {
                "_id" : -1
        },
        "new" : true
}
> mod.new=true
true
> mod
{
        "query" : {
                "elements" : 1
        },
        "update" : {
                "$set" : {
                        "touched" : true
                }
        },
        "sort" : {
                "_id" : -1
        },
        "new" : true
```

```
}

> mod.update.$set.touched=false

false

> mod

{

        "query" : {

                "elements" : 1

        },

        "update" : {

                "$set" : {

                        "touched" : false

                }

        },

        "sort" : {

                "_id" : -1

        },

        "new" : true

}


> db.b.findAndModify(mod)
```

{ "_id" : 3, "elements" : [1, 3], "touched" : false }

The above command returns the document after modification

Db.b.find()

{ "_id" : 1, "elements" : [1, 2, 3] }

{ "_id" : 2, "elements" : [2, 3] }

{ "_id" : 3, "elements" : [1, 3], "touched" : false }

{ "_id" : 4, "elements" : [3] }

To count the number of elements in the array

====================================

> db.b.find({"elements":{$size:2}})

{ "_id" : 2, "elements" : [2, 3] }

{ "_id" : 3, "elements" : [1, 3], "touched" : false }

> db.b.find({"elements":{$size:3}})

{ "_id" : 1, "elements" : [1, 2, 3] }

> db.b.find({"elements":{$size:1}})

{ "_id" : 4, "elements" : [3] }

Few Cursor Operations

=====================

```
> cur=db.people.find();null;

null

> cur.hasNext()

true

> cur.next()

{

        "_id" : ObjectId("573999bcb8bedb60cb42ff3f"),

        "name" : "akash",

        "title" : "Sr"

}

> cur.next()

{ "_id" : 1, "name" : "Hemanth", "title" : "Sr" }

> cur.next()

{ "_id" : 2, "name" : "mani", "title" : "Sr" }

> cur.next()

{ "_id" : 3, "name" : "vish", "depno" : 30, "title" : "Sr" }

> cur=db.people.find();null;

null

> cur.limit(3);null;

null
```

```
> while(cur.hasNext())printjson(cur.next())
{
        "_id" : ObjectId("573999bcb8bedb60cb42ff3f"),
        "name" : "akash",
        "title" : "Sr"
}
{ "_id" : 1, "name" : "Hemanth", "title" : "Sr" }
{ "_id" : 2, "name" : "mani", "title" : "Sr" }
```

MongoDB Aggregation Framework Basics Explained

To understand the MongoDB's aggregation framework, lets start with inserting the following data.

```
db.Student.insert ({Student_Name:"Kalki",Class:"2",Mark_Scored:100,Subject:["Tamil","E
nglish","Maths"]})

db.Student.insert ({Student_Name:"Matsya",Class:"1",Mark_Scored:10,Subject:["Tamil","E
nglish"]})



db.Student.insert ({Student_Name:"Krishna",Class:"1",Mark_Scored:50,Subject:["Tamil"]}
)

db.Student.insert ({Student_Name:"Buddha",Class:"2",Mark_Scored:60,Subject:["Tamil"]})

db.Student.insert ({Student_Name:"Rama",Class:"2",Mark_Scored:80,Subject:["Tamil"]})



db.Student.insert ({Student_Name:"Krishna",Class:"1",Mark_Scored:50,Subject:["English"
]})

db.Student.insert ({Student_Name:"Buddha",Class:"2",Mark_Scored:60,Subject:["English"]
})

db.Student.insert ({Student_Name:"Rama",Class:"2",Mark_Scored:80,Subject:["English"]})



db.Student.insert ({Student_Name:"Matsya",Class:"1",Mark_Scored:67,Subject:["Maths"]})

db.Student.insert ({Student_Name:"Krishna",Class:"1",Mark_Scored:95,Subject:["Maths"]}
)

db.Student.insert ({Student_Name:"Buddha",Class:"2",Mark_Scored:88,Subject:["Maths"]})

db.Student.insert ({Student_Name:"Rama",Class:"2",Mark_Scored:40,Subject:["Maths"]})
```

Pipeline

The aggregation framework is based on pipeline concept, just like unix pipeline. There can be N number of operators. Output of first operator will be fed as input to the second operator. Output of second operator will be fed as input to the third operator and so on.

Pipeline Operators

Following are the basic pipeline operators and let us make use of these operators over the sample data which we created.

$match

1. $unwind
2. $group
3. $project
4. $skip
5. $limit
6. $sort

$match

This is similar to MongoDB Collection's find method and SQL's WHERE clause. Basically this filters the data which is passed on to the next operator. There can be multiple $match operators in the pipeline.

Note: The data what we pass to the aggregate function should be a list of Javascript objects (Python dictionaries). Each and every operator should be in a separate javascript object like shown in all the examples below.

Example: We want to consider only the marks of the students who study in Class "2"

```
db.Student.aggregate ([
{
"$match":
{
"Class":"2"
}
}
])
```

and the result is

```
{
        "result":[
                {
                        "_id":ObjectId("517cbb98eccb9ee3d000fa5c"),
                        "Student_Name":"Kalki",
                "Class":"2",
                "Mark_Scored":100,
                "Subject":[
                        "Tamil",
                        "English",
                        "Maths"
                ]
        },
        {
                "_id":ObjectId("517cbb98eccb9ee3d000fa5f"),
                "Student_Name":"Buddha",
                "Class":"2",
                "Mark_Scored":60,
                "Subject":[
                        "Tamil"
                ]
        },
        {
                "_id":ObjectId("517cbb98eccb9ee3d000fa60"),
                "Student_Name":"Rama",
                "Class":"2",
                "Mark_Scored":80,
                "Subject":[
                        "Tamil"
                ]
        },
                {
                        "_id":ObjectId("517cbb98eccb9ee3d000fa62"),
                        "Student_Name":"Buddha",
                        "Class":"2",
                        "Mark_Scored":60,
                        "Subject":[
                                "English"
                        ]
                },
                {
                        "_id":ObjectId("517cbb98eccb9ee3d000fa63"),
                        "Student_Name":"Rama",
                        "Class":"2",
                        "Mark_Scored":80,
                        "Subject":[
                                "English"
                        ]
                },
                {
                        "_id":ObjectId("517cbb98eccb9ee3d000fa66"),
                        "Student_Name":"Buddha",
                        "Class":"2",
                        "Mark_Scored":88,
                        "Subject":[
                                "Maths"
                                ]
                        },
                        {
                                "_id":ObjectId("517cbb98eccb9ee3d000fa67"),
                                "Student_Name":"Rama",
                                "Class":"2",
                                "Mark_Scored":40,
                                "Subject":[
                                        "Maths"
                                ]
                        }
                ],
                "ok":1
}
```

Let us say, we want to consider only the marks of the students who study in Class "2" and whose marks are more than or equal to 80

```
db.Student.aggregate ([
{
"$match":
{
"Class":"2",
"Mark_Scored":
```

```
{
"$gte":80
}
}
}
])
```

Or we can use $match operator twice to achieve the same result

```
db.Student.aggregate ([
{
"$match":
{
"Class":"2",
}
},
{
"$match":
{
"Mark_Scored":
{
"$gte":80
}
}
}
])
```

and the result would be

```
{
        "result":[
                {
                        "_id":ObjectId("517cbb98eccb9ee3d000fa5c"),
                        "Student_Name":"Kalki",
                        "Class":"2",
                        "Mark_Scored":100,
                        "Subject":[
                                "Tamil",
                                "English",
                                "Maths"
                        ]
                },
                {
                        "_id":ObjectId("517cbb98eccb9ee3d000fa60"),
                        "Student_Name":"Rama",
                        "Class":"2",
                        "Mark_Scored":80,
                        "Subject":[
                                "Tamil"
                                ]
                        },
                        {
                                "_id":ObjectId("517cbb98eccb9ee3d000fa63"),
                                "Student_Name":"Rama",
                                "Class":"2",
                                "Mark_Scored":80,
                                "Subject":[
                                        "English"
                                ]
                        },
                        {
                                "_id":ObjectId("517cbb98eccb9ee3d000fa66"),
                                "Student_Name":"Buddha",
                                "Class":"2",
                                "Mark_Scored":88,
                                "Subject":[
                                        "Maths"
                                ]
                        }
                ],
                "ok":1
}
```

$unwind

This will be very useful when the data is stored as list. When the unwind operator is applied on a list data field, it will generate a new record for each and every element of the list data field on which unwind is applied. It basically flattens the data. Lets see an example to understand this better

Note: The field name, on which unwind is applied, should be prefixed with $ (dollar sign)

Example: Lets apply unwind over "Kalki"'s data.

```
db.Student.aggregate ([
{
"$match":
{
"Student_Name":"Kalki",
}
}
])
```

This generates the following output

```
{
        "result":[
                {
                        "_id":ObjectId("517cbb98eccb9ee3d000fa5c"),
                        "Student_Name":"Kalki",
                                "Class":"2",

                                "Mark_Scored":100,

                                "Subject":[

                                        "Tamil",

                                        "English",

                                        "Maths"

                                ]

                        }

                ],

                "ok":1

}
```

Whereas

```
db.Student.aggregate ([

{

"$match":

{

"Student_Name":"Kalki",

}

},

{

"$unwind":"$Subject"

}
])
```

will generate the following output

```
{
        "result":[
                {
                        "_id":ObjectId("517cbb98eccb9ee3d000fa5c"),
                        "Student_Name":"Kalki",
                        "Class":"2",
                        "Mark_Scored":100,
                        "Subject":"Tamil"
                },
                {
                        "_id":ObjectId("517cbb98eccb9ee3d000fa5c"),
                        "Student_Name":"Kalki",
                        "Class":"2",
                        "Mark_Scored":100,
                        "Subject":"English"
                },
                {
                        "_id":ObjectId("517cbb98eccb9ee3d000fa5c"),
                        "Student_Name":"Kalki",
                        "Class":"2",
                            "Mark_Scored":100,
                            "Subject":"Maths"
                    }
        ],
        "ok":1
}
```

$group

Now that we have flatten the data to be processed, lets try and group the data to process them. The group pipeline operator is similar to the SQL's GROUP BY clause. In SQL, we can't use GROUP BY unless we use any of the aggregation functions. The same way, we have to use an aggregation function in MongoDB as well. You can read more about the aggregation functions here. As most of them are like in SQL, I don't think much explanation would be needed.

Note: The _id element in group operator is a must. We cannot change it to some other name. MongoDB identifies the grouping expression with the _id field only.

Example: Lets try and get the sum of all the marks scored by each and every student, in Class "2"

```
db.Student.aggregate ([
{
"$match":
{
```

```
"Class":"2"

}

},

{

"$unwind":"$Subject"

},

{

"$group":

{

"_id":

{

"Student_Name":"$Student_Name"

},

"Total_Marks":

{

"$sum":"$Mark_Scored"

}

}

}

])
```

If we look at this aggregation example, we have specified an _id element and Total_Marks element. The _id element tells MongoDB to group the documents based on Student_Name field. The Total_Marks uses an aggregation function$sum, which basically adds up all the marks and returns the sum. This will produce this Output

```
{
        "result":[
                {
                        "_id":{
                                "Student_Name":"Rama"
                        },
                        "Total_Marks":200
                },
                {
                        "_id":{
                                "Student_Name":"Buddha"
                        },
                        "Total_Marks":208
                },
                {
                        "_id":{
                                "Student_Name":"Kalki"
                        },
                        "Total_Marks":300
                }
        ],
        "ok":1
}
```

We can use the sum function to count the number of records match the grouped data. Instead of "$sum": "$Mark_Scored", "$sum": 1 will count the number of records. "$sum": 2 will add 2 for each and every grouped data.

```
db.Student.aggregate ([
{
"$match":
{
"Class":"2"
}
},
{
"$unwind":"$Subject"
},
{
"$group":
{
"_id":
{
"Student_Name":"$Student_Name"
},
"Total_Marks":
{
"$sum":1
}

}

}

])
```

This will produce this Output

```
{
        "result":[
                {
                        "_id":{
                                "Student_Name":"Rama"
                        },
                        "Total_Marks":3
                },
                {
                        "_id":{
                                "Student_Name":"Buddha"
                        },
                        "Total_Marks":3
                },
                {
                        "_id":{
                                "Student_Name":"Kalki"
                        },
                         "Total_Marks":3
                 }
        ],
        "ok":1
}
```

This is because each and every student has marks for three subjects.

$project

The project operator is similar to SELECT in SQL. We can use this to rename the field names and select/deselect the fields to be returned, out of the grouped fields. If we specify 0 for a field, it will NOT be sent in the pipeline to the next operator. We can even flatten the data using project as shown in the example below

Example:

```
db.Student.aggregate ([
{
"$match":
{
"Class":"2"
}
},
{
```

```
"$unwind":"$Subject"

},

{

"$group":

{

"_id":

{

"Student_Name":"$Student_Name"

},

"Total_Marks":

{

"$sum":"$Mark_Scored"

}

}

},

{

"$project":

{

"_id":0,

"Name":"$_id.Student_Name",

"Total":"$Total_Marks"

}

}

])
```

will result in

```
{
        "result":[
                {
                        "Name":"Rama",
                        "Total":200
                },
                {
                        "Name":"Buddha",
                        "Total":208
                },
                {
                        "Name":"Kalki",
                        "Total":300
                }
        ],
        "ok":1
}
```

Lets say we try to retrieve Subject field by specifying project like shown below. MongoDB will simply ignore the Subject field, since it is not used in the group operator's _id field.

```
"$project":
```

```
{
"_id":0,
"Subject":1,
"Name":"$_id.Student_Name",
"Total":"$Total_Marks"
}
```

$sort

This is similar to SQL's ORDER BY clause. To sort a particular field in descending order specify -1 and specify 1 if that field has to be sorted in ascending order. I don't think this section needs more explanation. Lets straight away look at an example

Example:

```
db.Student.aggregate ([
{
"$match":
{
"Class":"2"
}
},
{
"$unwind":"$Subject"
},
{
```

```
"$group":

{

"_id":

{

"Student_Name":"$Student_Name"

},

"Total_Marks":

{

"$sum":"$Mark_Scored"

}

}

},

{

"$project":

{

"_id":0,

"Name":"$_id.Student_Name",

"Total":"$Total_Marks"

}

},

{

"$sort":

{

"Total":-1,

"Name":1
```

|

```
}
}
])
```

Will Sort the data based on Marks in descending Order first and then by Name in Ascending Order.

```
{
        "result":[
                {
                        "Name":"Kalki",
                        "Total":300
                },
                {
                        "Name":"Buddha",
                        "Total":208
                },
                {
                        "Name":"Rama",
                        "Total":200
                }
        ],
        "ok":1
}
```

$limit and $skip

These two operators can be used to limit the number of documents being returned. They will be more useful when we need pagination support.

Example:

```
db.Student.aggregate ([
{
"$match":
{
"Class":"2"
}
},
{
"$unwind":"$Subject"
},
{
"$group":
{
"_id":
{
"Student_Name":"$Student_Name"
},
"Total_Marks":
{
"$sum":"$Mark_Scored"
}
}
},
{
"$project":
{
"_id":0,
"Name":"$_id.Student_Name",
"Total":"$Total_Marks"
}
},
{
"$sort":
{
"Total":-1,
"Name":1
}
},
{
"$limit":2,
},
{
"$skip":1,
}
])
```

will result in

```
{"result":[{"Name":"Buddha","Total":208}],"ok":1}
```

Because the limit operator receives 3 documents from the sort operator and allows only the first two documents to pass through it, thereby dropping Rama's record. The skip operator skips one document (that means the first document (Kalki's document) is dropped) and allows only the Buddha's document to pass through.

All the examples shown here are readily usable with pyMongo (Just replace db.Student with your Collection object name)

Aggregation Framework Example

1. Import the json documents into a database called test and create a collection by name zips
   mongoimport –db test –collection zips –file D:\zips.json

Each document in this collection has the following form:

```
{
  "_id" : "35004",
  "city" : "Acmar",
  "state" : "AL",
  "pop" : 6055,
  "loc" : [-86.51557, 33.584132]
}
```

- The _id field holds the zipcode as a string.
- The city field holds the city name.
- The state field holds the two letter state abbreviation.
- The pop field holds the population.
- The loc field holds the location as a [latitude, longitude] array.

2. Find the states with Population over 10 Million

```
db.zips.aggregate([
  {"$group" => {_id: "$state", total_pop: {"$sum" => "$pop"}}},
  {"$match" => {total_pop: {"$gte" => 10_000_000}}}
])
```

3. Find the average city population by state

```
db.zips.aggregate([
  {"$group" => {_id: {state: "$state", city: "$city"}, pop: {"$sum" =>
"$pop"}}},
  {"$group" => {_id: "$_id.state", avg_city_pop: {"$avg" => "$pop"}}},
  {"$sort" => {avg_city_pop: -1}},
  {"$limit" => 3}
])
```

4. Find the largest and smalles cities by state

```
db.zips.aggregate([
  {"$group" => {_id: {state: "$state", city: "$city"}, pop: {"$sum" => "$pop"}}},
  {"$sort" => {pop: 1}},
  {"$group" => {
              _id: "$_id.state",
      smallest_city: {"$first" => "$_id.city"},
       smallest_pop: {"$first" => "$pop"},
       biggest_city: { "$last" => "$_id.city"},
        biggest_pop: { "$last" => "$pop"}
    }
  }
])
```

Creating Indexes

See below the steps to configure the WiredTiger Storage Engine

```
C:\Users\srsenthi>mkdir WT

A subdirectory or file WT already exists.


C:\Users\srsenthi>mongod --dbpath  WT --storageEngine
wiredTiger
```

Index   Creation Examples

1.

```
> for (i=0;i<1000000;i++) { db.users.insert({"i" : i,"username" : "user" + i,
"age":Math.floor(Math.random()*120),"created" : new Date()});}

WriteResult({ "nInserted" : 1 })

> db.users.find().count()

1000000

> db.users.find({username:"user01"}).explain()

{

        "queryPlanner" : {
                "plannerVersion" : 1,
                "namespace" : "test.users",
                "indexFilterSet" : false,
                "parsedQuery" : {
                        "username" : {
                                "$eq" : "user01"
                        }
                },
                "winningPlan" : {
                        "stage" : "COLLSCAN",
                        "filter" : {
                                "username" : {
                                        "$eq" : "user01"
                                }
                        },
                        "direction" : "forward"
                },
                "rejectedPlans" : [ ]
        },
        "serverInfo" : {
```

```
		"host" : "LIN73000604",
		"port" : 27017,
		"version" : "3.2.4",
		"gitVersion" : "e2ee9ffcf9f5a94fad76802e28cc978718bb7a30"
	},
	"ok" : 1
}
```

3. > db.users.find({username: "user101"}).limit(1).explain()

```
{
	"queryPlanner" : {
		"plannerVersion" : 1,
		"namespace" : "test.users",
		"indexFilterSet" : false,
		"parsedQuery" : {
			"username" : {
				"$eq" : "user101"
			}
		},
		"winningPlan" : {
			"stage" : "LIMIT",
                "limitAmount" : 1,
                "inputStage" : {
                        "stage" : "COLLSCAN",
                        "filter" : {
                                "username" : {
                                        "$eq" : "user101"
                                }
                        },
                        "direction" : "forward"
                }
        },
        "rejectedPlans" : [ ]
},
"serverInfo" : {
        "host" : "LIN73000604",
        "port" : 27017,
        "version" : "3.2.4",
        "gitVersion" : "e2ee9ffcf9f5a94fad76802e28cc978718bb7a30"
},
"ok" : 1
```

}

4. > db.users.ensureIndex({"username" : 1}) (creating Single key index)

```
{
        "createdCollectionAutomatically" : false,
        "numIndexesBefore" : 1,
        "numIndexesAfter" : 2,
        "ok" : 1
}
```

Test for Index usage :

=====================

5. > db.users.find({"username" : "user101"}).explain()

```
{
        "queryPlanner" : {
                "plannerVersion" : 1,
                "namespace" : "test.users",
                "indexFilterSet" : false,
                "parsedQuery" : {
                        "username" : {
                                "$eq" : "user101"
                        }
```

```
},
"winningPlan" : {
        "stage" : "FETCH",
        "inputStage" : {
                "stage" : "IXSCAN",
                "keyPattern" : {
                        "username" : 1
                },
                "indexName" : "username_1",
                "isMultiKey" : false,
                "isUnique" : false,
                "isSparse" : false,
                "isPartial" : false,
                "indexVersion" : 1,
                "direction" : "forward",
                "indexBounds" : {
                        "username" : [
                                "[\"user101\", \"user101\"]"
                        ]
                }
```

```
                        }
                },
                "rejectedPlans" : [ ]
        },
        "serverInfo" : {
                "host" : "LIN73000604",
                "port" : 27017,
                "version" : "3.2.4",
                "gitVersion" : "e2ee9ffcf9f5a94fad76802e28cc978718bb7a30"
        },
        "ok" : 1
}
```

6. Sorting based on age and username

```
> db.users.find().sort({"age" : 1, "username" : 1})
Error: error: {
        "waitedMS" : NumberLong(0),
        "ok" : 0,
        "errmsg" : "Executor error during find command: OperationFailed: Sort
operation used more than the maximum 33554432 bytes of RAM. Add an index,
or specify a smaller limit."

,

        "code" : 96

}
```

To optimize the sort ,we could make an index on username and age called as compound index

```
> db.users.ensureIndex({"age" : 1, "username" : 1})

{

        "createdCollectionAutomatically" : false,

        "numIndexesBefore" : 2,

        "numIndexesAfter" : 3,

        "ok" : 1

}
```

This is called a *compound index* and is useful if your query has multiple sort directions
or multiple keys in the criteria. A compound index is an index on more than one field

Each index entry contains an age and a username and points to the location of a document
on disk (represented by the hexadecimal numbers, which can be ignored)

```
> db.users.find({"age" : 21}).sort({"username" : -1})
{ "_id" : ObjectId("5736a5849bbb29d3e0ca8207"), "i" : 999978, "username" : "user999978", "age" : 21, "created" : ISODate("2016-05-14T04:11:48.550Z") }
{ "_id" : ObjectId("5736a5849bbb29d3e0ca81d0"), "i" : 999923, "username" : "user999923", "age" : 21, "created" : ISODate("2016-05-14T04:11:48.436Z") }
{ "_id" : ObjectId("5736a5849bbb29d3e0ca8137"), "i" : 999770, "username" : "user999770", "age" : 21, "created" : ISODate("2016-05-14T04:11:48.112Z") }
{ "_id" : ObjectId("5736a5839bbb29d3e0ca80f2"), "i" : 999701, "username" : "user999701", "age" : 21, "created" : ISODate("2016-05-14T04:11:47.968Z") }
{ "_id" : ObjectId("5736a5839bbb29d3e0ca8026"), "i" : 999497, "username" : "user999497", "age" : 21, "created" : ISODate("2016-05-14T04:11:47.539Z") }
{ "_id" : ObjectId("5736a5839bbb29d3e0ca7f28"), "i" : 999243, "username" : "user999243", "age" : 21, "created" : ISODate("2016-05-14T04:11:47.001Z") }
{ "_id" : ObjectId("5736a5829bbb29d3e0ca7ec1"), "i" : 999140, "username" : "user999140", "age" : 21, "created" : ISODate("2016-05-14T04:11:46.784Z") }
{ "_id" : ObjectId("5736a5829bbb29d3e0ca7eb8"), "i" : 999131, "username" : "user999131", "age" : 21, "created" : ISODate("2016-05-14T04:11:46.758Z") }
{ "_id" : ObjectId("5736a5829bbb29d3e0ca7ea0"), "i" : 999107, "username" : "user999107", "age" : 21, "created" : ISODate("2016-05-14T04:11:46.697Z") }
{ "_id" : ObjectId("5736a5829bbb29d3e0ca7da8"), "i" : 998859, "username" : "user998859", "age" : 21, "created" : ISODate("2016-05-14T04:11:46.164Z") }
{ "_id" : ObjectId("5736a5829bbb29d3e0ca7da0"), "i" : 998851, "username" : "user998851", "age" : 21, "created" : ISODate("2016-05-14T04:11:46.147Z") }
{ "_id" : ObjectId("5736a5809bbb29d3e0ca7b36"), "i" : 998233, "username" :
"user998233", "age" : 21, "created" : ISODate("2016-05-14T04:11:44.840Z") }

{ "_id" : ObjectId("5736a5809bbb29d3e0ca7a83"), "i" : 998054, "username" :
"user998054", "age" : 21, "created" : ISODate("2016-05-14T04:11:44.467Z") }

{ "_id" : ObjectId("5736a5809bbb29d3e0ca7a65"), "i" : 998024, "username" :
"user998024", "age" : 21, "created" : ISODate("2016-05-14T04:11:44.407Z") }

{ "_id" : ObjectId("5736a5809bbb29d3e0ca7a4c"), "i" : 997999, "username" :
"user997999", "age" : 21, "created" : ISODate("2016-05-14T04:11:44.354Z") }

{ "_id" : ObjectId("5736a5809bbb29d3e0ca7a2d"), "i" : 997968, "username" :
"user997968", "age" : 21, "created" : ISODate("2016-05-14T04:11:44.285Z") }

{ "_id" : ObjectId("5736a57e9bbb29d3e0ca7752"), "i" : 997237, "username" :
"user997237", "age" : 21, "created" : ISODate("2016-05-14T04:11:42.719Z") }

{ "_id" : ObjectId("57369d8d9bbb29d3e0bcc54e"), "i" : 99697, "username" :
"user99697", "age" : 21, "created" : ISODate("2016-05-14T03:37:49.482Z") }

{ "_id" : ObjectId("5736a57d9bbb29d3e0ca7608"), "i" : 996907, "username" :
"user996907", "age" : 21, "created" : ISODate("2016-05-14T04:11:41.938Z") }

{ "_id" : ObjectId("5736a57c9bbb29d3e0ca73cb"), "i" : 996334, "username" :
"user996334", "age" : 21, "created" : ISODate("2016-05-14T04:11:40.684Z") }
```

**MongoDB can jump directly to the correct age
and doesn’t need to sort the results as traversing the index returns the data in the
correct order.**

```
db.users.find({"age" : {"$gte" : 21, "$lte" : 30}})
```

This is a multi-value query, which looks for documents matching multiple values (in this case, all ages between 21 and 30).

```
{ "_id" : ObjectId("57369d8e9bbb29d3e0bcc68a"), "i" : 100013, "username" :
"user100013", "age" : 21, "created" : ISODate("2016-05-14T03:37:50.292Z") }

{ "_id" : ObjectId("57369d8e9bbb29d3e0bcc691"), "i" : 100020, "username" :
"user100020", "age" : 21, "created" : ISODate("2016-05-14T03:37:50.307Z") }

{ "_id" : ObjectId("57369d8e9bbb29d3e0bcc6bc"), "i" : 100063, "username" :
"user100063", "age" : 21, "created" : ISODate("2016-05-14T03:37:50.414Z") }

{ "_id" : ObjectId("57369d8e9bbb29d3e0bcc744"), "i" : 100199, "username" :
"user100199", "age" : 21, "created" : ISODate("2016-05-14T03:37:50.706Z") }

{ "_id" : ObjectId("57369d8e9bbb29d3e0bcc79f"), "i" : 100290, "username" :
"user100290", "age" : 21, "created" : ISODate("2016-05-14T03:37:50.912Z") }

{ "_id" : ObjectId("57369d8f9bbb29d3e0bcc816"), "i" : 100409, "username" :
"user100409", "age" : 21, "created" : ISODate("2016-05-14T03:37:51.181Z") }

{ "_id" : ObjectId("57369d8f9bbb29d3e0bcc8a2"), "i" : 100549, "username" :
"user100549", "age" : 21, "created" : ISODate("2016-05-14T03:37:51.474Z") }

{ "_id" : ObjectId("57369cc69bbb29d3e0bb6748"), "i" : 10091, "username" :
"user10091", "age" : 21, "created" : ISODate("2016-05-14T03:34:30.909Z") }

{ "_id" : ObjectId("57369d909bbb29d3e0bcca5b"), "i" : 100990, "username" :
"user100990", "age" : 21, "created" : ISODate("2016-05-14T03:37:52.493Z") }

{ "_id" : ObjectId("57369d909bbb29d3e0bcca86"), "i" : 101033, "username" :
"user101033", "age" : 21, "created" : ISODate("2016-05-14T03:37:52.592Z") }
```

```
{ "_id" : ObjectId("57369d909bbb29d3e0bccaf9"), "i" : 101148, "username" :
"user101148", "age" : 21, "created" : ISODate("2016-05-14T03:37:52.868Z") }

{ "_id" : ObjectId("57369d919bbb29d3e0bccbeb"), "i" : 101390, "username" :
"user101390", "age" : 21, "created" : ISODate("2016-05-14T03:37:53.430Z") }

{ "_id" : ObjectId("57369d919bbb29d3e0bccc37"), "i" : 101466, "username" :
"user101466", "age" : 21, "created" : ISODate("2016-05-14T03:37:53.593Z") }

{ "_id" : ObjectId("57369cc79bbb29d3e0bb6793"), "i" : 10166, "username" :
"user10166", "age" : 21, "created" : ISODate("2016-05-14T03:34:31.067Z") }

{ "_id" : ObjectId("57369d929bbb29d3e0bccdc0"), "i" : 101859, "username" :
"user101859", "age" : 21, "created" : ISODate("2016-05-14T03:37:54.489Z") }

{ "_id" : ObjectId("57369d929bbb29d3e0bccdd1"), "i" : 101876, "username" :
"user101876", "age" : 21, "created" : ISODate("2016-05-14T03:37:54.535Z") }

{ "_id" : ObjectId("57369d929bbb29d3e0bccdf2"), "i" : 101909, "username" :
"user101909", "age" : 21, "created" : ISODate("2016-05-14T03:37:54.604Z") }

{ "_id" : ObjectId("57369d929bbb29d3e0bcce17"), "i" : 101946, "username" :
"user101946", "age" : 21, "created" : ISODate("2016-05-14T03:37:54.680Z") }

{ "_id" : ObjectId("57369d929bbb29d3e0bcce62"), "i" : 102021, "username" :
"user102021", "age" : 21, "created" : ISODate("2016-05-14T03:37:54.839Z") }

{ "_id" : ObjectId("57369d929bbb29d3e0bcce75"), "i" : 102040, "username" :
"user102040", "age" : 21, "created" : ISODate("2016-05-14T03:37:54.879Z") }

Type "it" for more

> db.users.find({"age" : {"$gte" : 21, "$lte" : 30}}).sort({"username" :

... 1})
```

```
{ "_id" : ObjectId("57369d8e9bbb29d3e0bcc68a"), "i" : 100013, "username" :
"user100013", "age" : 21, "created" : ISODate("2016-05-14T03:37:50.292Z") }

{ "_id" : ObjectId("57369d8e9bbb29d3e0bcc691"), "i" : 100020, "username" :
"user100020", "age" : 21, "created" : ISODate("2016-05-14T03:37:50.307Z") }

{ "_id" : ObjectId("57369d8e9bbb29d3e0bcc6a9"), "i" : 100044, "username" :
"user100044", "age" : 26, "created" : ISODate("2016-05-14T03:37:50.365Z") }

{ "_id" : ObjectId("57369cc69bbb29d3e0bb66f3"), "i" : 10006, "username" :
"user10006", "age" : 26, "created" : ISODate("2016-05-14T03:34:30.728Z") }

{ "_id" : ObjectId("57369d8e9bbb29d3e0bcc6bc"), "i" : 100063, "username" :
"user100063", "age" : 21, "created" : ISODate("2016-05-14T03:37:50.414Z") }

{ "_id" : ObjectId("57369d8e9bbb29d3e0bcc6ce"), "i" : 100081, "username" :
"user100081", "age" : 22, "created" : ISODate("2016-05-14T03:37:50.454Z") }

{ "_id" : ObjectId("57369d8e9bbb29d3e0bcc6d1"), "i" : 100084, "username" :
"user100084", "age" : 22, "created" : ISODate("2016-05-14T03:37:50.461Z") }

{ "_id" : ObjectId("57369d8e9bbb29d3e0bcc6da"), "i" : 100093, "username" :
"user100093", "age" : 28, "created" : ISODate("2016-05-14T03:37:50.479Z") }

{ "_id" : ObjectId("57369d8e9bbb29d3e0bcc6db"), "i" : 100094, "username" :
"user100094", "age" : 28, "created" : ISODate("2016-05-14T03:37:50.481Z") }

{ "_id" : ObjectId("57369d8e9bbb29d3e0bcc6df"), "i" : 100098, "username" :
"user100098", "age" : 23, "created" : ISODate("2016-05-14T03:37:50.490Z") }

{ "_id" : ObjectId("57369d8e9bbb29d3e0bcc6e0"), "i" : 100099, "username" :
"user100099", "age" : 23, "created" : ISODate("2016-05-14T03:37:50.492Z") }

{ "_id" : ObjectId("57369d8e9bbb29d3e0bcc6e2"), "i" : 100101, "username" :
"user100101", "age" : 22, "created" : ISODate("2016-05-14T03:37:50.497Z") }
```

```
{ "_id" : ObjectId("57369d8e9bbb29d3e0bcc6e6"), "i" : 100105, "username" :
"user100105", "age" : 26, "created" : ISODate("2016-05-14T03:37:50.506Z") }

{ "_id" : ObjectId("57369d8e9bbb29d3e0bcc719"), "i" : 100156, "username" :
"user100156", "age" : 23, "created" : ISODate("2016-05-14T03:37:50.611Z") }

{ "_id" : ObjectId("57369d8e9bbb29d3e0bcc71b"), "i" : 100158, "username" :
"user100158", "age" : 30, "created" : ISODate("2016-05-14T03:37:50.615Z") }

{ "_id" : ObjectId("57369d8e9bbb29d3e0bcc73d"), "i" : 100192, "username" :
"user100192", "age" : 25, "created" : ISODate("2016-05-14T03:37:50.687Z") }

{ "_id" : ObjectId("57369d8e9bbb29d3e0bcc744"), "i" : 100199, "username" :
"user100199", "age" : 21, "created" : ISODate("2016-05-14T03:37:50.706Z") }

{ "_id" : ObjectId("57369cc69bbb29d3e0bb6701"), "i" : 10020, "username" :
"user10020", "age" : 29, "created" : ISODate("2016-05-14T03:34:30.758Z") }

{ "_id" : ObjectId("57369d8e9bbb29d3e0bcc74b"), "i" : 100206, "username" :
"user100206", "age" : 27, "created" : ISODate("2016-05-14T03:37:50.722Z") }

{ "_id" : ObjectId("57369d8e9bbb29d3e0bcc74c"), "i" : 100207, "username" :
"user100207", "age" : 29, "created" : ISODate("2016-05-14T03:37:50.723Z") }
```

```
db.users.find({"age" : {"$gte" : 21, "$lte" : 30}}).sort({"username" :

1})
```

However, the index doesn’t return the usernames in sorted order and the query

requested that the results be sorted by username, so MongoDB has to sort the results

in memory before returning them. Thus, this query is usually less efficient than the

queries above.

One other index you can use in the last example is the same keys in reverse order:
{"username" : 1, "age" : 1}. MongoDB will then traverse all the index entries, but
in the order you want them back in. It will pick out the matching documents using the
"age" part of the index:

This is good in that it does not require any giant in-memory sorts. However, it does
have to scan the entire index to find all matches. Thus, putting the sort key first is
generally a good strategy when you' re using a limit so MongoDB can stop scanning the
index after a couple of matches.

```
> db.users.find({"age" : {"$gte" : 21, "$lte" : 30}}).sort({"username" :
1}).explain()

{

    "queryPlanner" : {

        "plannerVersion" : 1,

        "namespace" : "test.users",

        "indexFilterSet" : false,
"parsedQuery" : {
        "$and" : [
                {
                        "age" : {
                                "$lte" : 30
                        }
                },
                {
                        "age" : {
                                "$gte" : 21
                        }
                }
        ]
},
"winningPlan" : {
        "stage" : "FETCH",
        "filter" : {
                "$and" : [
                        {
                                "age" : {
```

```
                        "$lte" : 30
                    }
                },
                {
                    "age" : {
                        "$gte" : 21
                    }
                }
            ]
        },
        "inputStage" : {
            "stage" : "IXSCAN",
            "keyPattern" : {
                "username" : 1
            },
            "indexName" : "username_1",
            "isMultiKey" : false,
            "isUnique" : false,
            "isSparse" : false,
            "isPartial" : false,
                "indexVersion" : 1,
                "direction" : "forward",
                "indexBounds" : {
                    "username" : [
                        "[MinKey, MaxKey]"
                    ]
                }
            }
        },
        "rejectedPlans" : [
            {
                "stage" : "SORT",
                "sortPattern" : {
                    "username" : 1
                },
                "inputStage" : {
                    "stage" : "SORT_KEY_GENERATOR",
                    "inputStage" : {
                        "stage" : "FETCH",
                        "inputStage" : {
```

```
                    "stage" : "IXSCAN",
                    "keyPattern" : {
                        "age" : 1,
                        "username" : 1
                    },
"indexName" : "age_1_username_1",
                    "isMultiKey" : false,
                    "isUnique" : false,
                    "isSparse" : false,
                    "isPartial" : false,
                    "indexVersion" : 1,
                    "direction" : "forward",
                    "indexBounds" : {
                        "age" : [
                            "[21.0, 30.0]"
                        ],
                        "username" : [
                            "[MinKey, MaxKey]"
                        ]
                    }
```

```
                                        }
                                    }
                                }
                            }
                        ]
                },
                "serverInfo" : {
                        "host" : "LIN73000604",
                        "port" : 27017,
                        "version" : "3.2.4",
                        "gitVersion" : "e2ee9ffcf9f5a94fad76802e28cc978718bb7a30"
                },
                "ok" : 1
}
> db.users.dropIndex({"age" : 1, "username" : 1})
{ "nIndexesWas" : 3, "ok" : 1 }
```

> db.users.ensureIndex({"username" : 1, "age" : 1}) (Compound index created with the sort key as the prefix)

This avoids memory level sorts … but scans the entire index documents

```
{
        "createdCollectionAutomatically" : false,
        "numIndexesBefore" : 2,
        "numIndexesAfter" : 3,
        "ok" : 1
}
> db.users.find({"age" : {"$gte" : 21, "$lte" : 30}}).sort({"username" :
1}).explain()
{
        "queryPlanner" : {
                "plannerVersion" : 1,
                "namespace" : "test.users",
                "indexFilterSet" : false,
                "parsedQuery" : {
                        "$and" : [
                                {
                                        "age" : {
                                                "$lte" : 30
                                        }
                                },
                {
                    "age" : {
                        "$gte" : 21
                    }
                }
            ]
        },
        "winningPlan" : {
            "stage" : "FETCH",
            "filter" : {
                "$and" : [
                    {
                        "age" : {
                            "$lte" : 30
                        }
                    },
                    {
                        "age" : {
                            "$gte" : 21
                        }
                }
            ]
        },
        "inputStage" : {
"stage" : "IXSCAN",
            "keyPattern" : {
                "username" : 1
            },
            "indexName" : "username_1",
            "isMultiKey" : false,
            "isUnique" : false,
            "isSparse" : false,
            "isPartial" : false,
            "indexVersion" : 1,
            "direction" : "forward",
            "indexBounds" : {
                "username" : [
                    "[MinKey, MaxKey]"
                ]
            }
```

```
                }
            },
            "rejectedPlans" : [ ]
        },
        "serverInfo" : {
            "host" : "LIN73000604",
            "port" : 27017,
            "version" : "3.2.4",
            "gitVersion" : "e2ee9ffcf9f5a94fad76802e28cc978718bb7a30"
        },
        "ok" : 1
}
```

Note : The index pattern of {"sortKey" : 1, "queryCriteria" : 1} often works well in applications, as most application do not want all possible results for a query but only the first few.

Putting the sort key first is generally a good strategy when you' re using a limit so MongoDB can stop scanning the index after a couple of matches.

Indexes are basically trees, with the smallest value on the leftmost leaf and the greatest on the rightmostrightmost. **If you have a "sortKey" that is a date (or any value that increases over time) then as you traverse the tree from left to right, you' re basically travelling forward in**

time. Thus, for applications that tend to use recent data more than older data, MongoDB only has to keep the rightmost (most recent) branches of the tree in memory, not the whole thing. An index like this is called right-balanced and, whenever possible, you should make your indexes right-balanced. The "_id" index is an example of a rightbalanced index.

To optimize compound sorts in different directions, use an index with matching directions.
In this example, we could use {"age" : 1, "username" : -1}

If our application also needed to optimize sorting by {"age" : 1, "username" : 1}, we would have to create a second index with those directions. To figure out which
directions to use for an index, simply match the directions your sort is using. Note that
inverse indexes (multiplying each direction by -1) are equivalent: {"age" : 1, "user
name" : -1} suits the same queries that {"age" : -1, "username" : 1} does.
Index direction only really matters when you' re sorting based on multiple criteria. If
you' re only sorting by a single key, MongoDB can just as easily read the index in the
opposite order. For example, if you had a sort on {"age" : -1} and an index on {"age" : 1}, MongoDB could optimize it just as well as if you had an index on {"age" :
-1} (so don' t create both!). The direction only matters for multikey sorts.

Types of Indexes

Unique Indexes :

> db.users.dropIndexes() (to drop the existing indexes)
{

```
	"nIndexesWas" : 2,
	"msg" : "non-_id indexes dropped for collection",
	"ok" : 1
}
> db.users.getIndexes()
[
	{
		"v" : 1,
		"key" : {
			"_id" : 1
		},
		"name" : "_id_",
		"ns" : "test.users"
	}
]
> db.users.ensureIndex({"username" : 1}, {"unique" : true})
{
	"createdCollectionAutomatically" : false,
	"numIndexesBefore" : 1,
	"numIndexesAfter" : 2,
	"ok" : 1
}
> db.users.insert({username: "bob"})
WriteResult({ "nInserted" : 1 })
> db.users.insert({username: "bob"})
WriteResult({
	"nInserted" : 0,
	"writeError" : {
		"code" : 11000,
		"errmsg" : "E11000 duplicate key error collection: test.users index:
username_1 dup key: { : \"bob\" }"
  }
})
```

If a key does not exist, the index stores its value as null for that document. This means that if you create a unique index and try to insert more than one document that is missing the indexed field, the inserts will fail because you already have a document with a value of null

```
> db.users.insert({age:23})
WriteResult({ "nInserted" : 1 })
> db.users.insert({"age":44})
WriteResult({
	"nInserted" : 0,
	"writeError" : {
		"code" : 11000,
		"errmsg" : "E11000 duplicate key error collection: test.users index:
username_1 dup key: { : null }"
	}
})
```

Note:

All fields must be smaller than 1024 bytes to be included in an index. MongoDB does not return any sort of error or warning if a document's fields cannot be indexed due to size. This means that keys longer than 8 KB will not be subject to the unique index constraints: you can insert identical 8 KB strings,

Compound Unique Indexes :

```
> db.users.ensureIndex({"username" : 1, "age" : 1}, {"unique" : true})
{
	"createdCollectionAutomatically" : false,
	"numIndexesBefore" : 1,
	"numIndexesAfter" : 2,
	"ok" : 1
}
> db.users.insert({"username" : "bob"})
WriteResult({
        "nInserted" : 0,
        "writeError" : {
                "code" : 11000,
                "errmsg" : "E11000 duplicate key error collection: test.users index:
username_1_age_1 dup key: { : \"bob\", : null }"
        }
})
>> db.users.insert({"username" : "bob", "age" : 23})
2016-05-14T13:56:24.904+0530 E QUERY    [thread1] SyntaxError: expected
expression, got '>' @(shell):1:0

>> db.users.insert({"username" : "fred", "age" : 23})
2016-05-14T13:56:37.909+0530 E QUERY    [thread1] SyntaxError: expected
expression, got '>' @(shell):1:0

>  db.users.insert({"username" : "fred", "age" : 23})
WriteResult({ "nInserted" : 1 })
> db.users.insert({"username" : "fred", "age" : 23})
WriteResult({
        "nInserted" : 0,
        "writeError" : {
                "code" : 11000,
                "errmsg" : "E11000 duplicate key error collection: test.users index:
username_1_age_1 dup key: { : \"fred\", : 23.0 }"
        }
})
```

Dropping Duplicates.
==================

To delete documents with duplicate values

″dropDups" option will save the first document found and remove any subsequent documents with duplicate values:

```
> db.people.ensureIndex({"username" : 1}, {"unique" : true, "dropDups" : true})
```

you have no control over which documents are dropped and which are kept (and MongoDB gives you no indication of which documents were dropped, if any). If your data is of any importance, do not use "dropDups".

```
> db.users.insert({"salary":1300,"eno":102})
WriteResult({ "nInserted" : 1 })
> db.users.insert({"salary":1400,"eno":104})
WriteResult({
        "nInserted" : 0,
        "writeError" : {
                "code" : 11000,
                "errmsg" : "E11000 duplicate key error collection: test.users index:
username_1_age_1 dup key: { : null, : null }"
        }
})
> db.users.dropIndexes()
{
        "nIndexesWas" : 2,
        "msg" : "non-_id indexes dropped for collection",
        "ok" : 1
}
```

In the above case , not more than one document can be entered with the index key as null. To avoid this we go for Sparse Indexes

Creating Sparse Indexes

unique indexes count null as a value, so you cannot have a unique index with more than one document missing the key. However, there are lots of cases where you may want the unique index to be enforced only if the key exists.
If you have a field that may or may not exist but must be unique when it does, you can
combine the unique option with the sparse option.

```
> db.users.ensureIndex({"username" : 1, "age" : 1}, {"unique" :
true,"sparse":true})
{
        "createdCollectionAutomatically" : false,
        "numIndexesBefore" : 1,
        "numIndexesAfter" : 2,
        "ok" : 1
}
> db.users.insert({"salary":1400,"eno":104})
WriteResult({ "nInserted" : 1 })
> db.users.insert({"salary":1300,"eno":103})
WriteResult({ "nInserted" : 1 })
```

Sparse indexes do not necessarily have to be unique. To make a non-unique sparse index,
simply do not include the unique option

Multikey indexes

db.foo.insert({a:1,b:2}) --- a document been created

```
> db.foo.find()
{ "_id" : ObjectId("56f0d67ee8436b9274fa4e80"), "a" : 1, "b" : 2 }
```

- Lets say we have to create index on a,b in the ascending order
- Db.foo.createIndex({a:1,b:1})
- To get more info on the usage of index ,we can go for explain method

Db.foo.explain().find({a:1,b:1})   in the query planner it tells about the winning plan and in the winning plan it says about the index scan it says that it is not a multikey index

Now in the next example lets create a document as the case given below

Db.foo.insert({a:3,b:[3,5,7]})

```
> db.multi.insert({a:1,b:2})
WriteResult({ "nInserted" : 1 })
> db.multi.find()
{ "_id" : ObjectId("57371bbe9bbb29d3e0ca822a"), "a" : 1, "b" : 2 }
> db.multi.find().explain()
{
	"queryPlanner" : {
		"plannerVersion" : 1,
		"namespace" : "test.multi",
		"indexFilterSet" : false,
		"parsedQuery" : {
			"$and" : [ ]
		},
		"winningPlan" : {
			"stage" : "COLLSCAN",
                "filter" : {
                    "$and" : [ ]
                },
                "direction" : "forward"
            },
            "rejectedPlans" : [ ]
        },
        "serverInfo" : {
            "host" : "LIN73000604",
            "port" : 27017,
            "version" : "3.2.4",
            "gitVersion" : "e2ee9ffcf9f5a94fad76802e28cc978718bb7a30"
        },
        "ok" : 1
}
> db.multi.createIndex({a:1,b:1})
{
        "createdCollectionAutomatically" : false,
        "numIndexesBefore" : 1,
        "numIndexesAfter" : 2,
        "ok" : 1
}
> db.multi.find().explain()
{
        "queryPlanner" : {
            "plannerVersion" : 1,
            "namespace" : "test.multi",
            "indexFilterSet" : false,
            "parsedQuery" : {
                "$and" : [ ]
            },
            "winningPlan" : {
                "stage" : "COLLSCAN",
                "filter" : {
                        "$and" : [ ]
                    },
                    "direction" : "forward"
                },
                "rejectedPlans" : [ ]
        },
        "serverInfo" : {
                "host" : "LIN73000604",
                "port" : 27017,
                "version" : "3.2.4",
                "gitVersion" : "e2ee9ffcf9f5a94fad76802e28cc978718bb7a30"
        },
        "ok" : 1
}
> db.multi.insert({a:3,b:4})
WriteResult({ "nInserted" : 1 })
> db.multi.find({a:1}).explain()
{
        "queryPlanner" : {
                "plannerVersion" : 1,
                "namespace" : "test.multi",
                "indexFilterSet" : false,
                "parsedQuery" : {
                        "a" : {
                                "$eq" : 1
                        }
                },
                "winningPlan" : {
                        "stage" : "FETCH",
                        "inputStage" : {
                                "stage" : "IXSCAN",
                                "keyPattern" : {
                                        "a" : 1,
                                        "b" : 1
                    },
                    "indexName" : "a_1_b_1",
                    "isMultiKey" : false,
                    "isUnique" : false,
                    "isSparse" : false,
                    "isPartial" : false,
                    "indexVersion" : 1,
                    "direction" : "forward",
                    "indexBounds" : {
                        "a" : [
                            "[1.0, 1.0]"
                        ],
                        "b" : [
                            "[MinKey, MaxKey]"
                        ]
                    }
                }
            },
            "rejectedPlans" : [ ]
        },
        "serverInfo" : {
            "host" : "LIN73000604",
            "port" : 27017,
            "version" : "3.2.4",
            "gitVersion" : "e2ee9ffcf9f5a94fad76802e28cc978718bb7a30"
        },
        "ok" : 1
}
> db.multi.insert({a:5,b:[1,2,3]})
WriteResult({ "nInserted" : 1 })
> db.multi.find({a:5}).explain()
{
        "queryPlanner" : {
            "plannerVersion" : 1,
        "namespace" : "test.multi",
        "indexFilterSet" : false,
        "parsedQuery" : {
            "a" : {
                "$eq" : 5
            }
        },
        "winningPlan" : {
            "stage" : "FETCH",
            "inputStage" : {
                "stage" : "IXSCAN",
                "keyPattern" : {
                    "a" : 1,
                    "b" : 1
                },
                "indexName" : "a_1_b_1",
"isMultiKey" : true,   ( becos  a is scalar and b is array) it uses multikey array
                "isUnique" : false,
                "isSparse" : false,
                "isPartial" : false,
                "indexVersion" : 1,
                "direction" : "forward",
                "indexBounds" : {
                    "a" : [
                        "[5.0, 5.0]"
                    ],
                    "b" : [
                        "[MinKey, MaxKey]"
                    ]
                }
            }
        },
        "rejectedPlans" : [ ]
    },
	"serverInfo" : {
		"host" : "LIN73000604",
		"port" : 27017,
		"version" : "3.2.4",
		"gitVersion" : "e2ee9ffcf9f5a94fad76802e28cc978718bb7a30"
	},
	"ok" : 1
}
> db.multi.insert({a:[2,3,4],b:[1,2,3]})    In a multi key index we cannot have
both a and b as arrays
WriteResult({
	"nInserted" : 0,
	"writeError" : {
		"code" : 10088,
		"errmsg" : "cannot index parallel arrays [b] [a]"
	}
})
```

Multikey with dot notation

We shall see how to use dot notation to reach deep into a document and add index that is something in a subdocument of the main document .

Examples

Db.students.findOne()

We have student_id and scores array that has a bunch of documents as elements in the array where each document has a type exame and a score .

And it s also a class id .

Let say we have to index on the score itself .

Db.students.createIndex([‘scores.score’:1]); -→ use square bracket for index creation as shown
It would take 15 or 20 mins to create this index
There are 10 million documents

If we use getIndexes ,we can see that there are two indexes one is on id and the other is on

scores.score which is the multi key index

To search of the records where any score is above the given value ,

The belo ex find s everything where scores.score is > 99

Db.students.explain.find({‘scores.score’:{‘gt”:99}})

In the explain plan we can see the index scan ,Winning plan included scores.score index with the scores.score between 99.0 and infinity ..

To find people that had exam score that was above 99 .

Db.students.explain().find({‘scores’:{$elemMatch:{type :’exam’,score:{‘gt’:99.8}}}});

Trying to inspect scores array inside the document .

Then we want ti find a document which has an element of array that is of type exam and a score .

LM match matches the document that contains an array field with at least one element that matches all specific criteria .

In other words , there might be more than one element , there might be more than one exam in this array that matches this criteria but . we ensure that we match atleast one with all this criteria .

So we are looking for element of type exam and score greater than 99.8

So in the result we could see that there is an exam score more than 99.8

Db.students.explain().find({‘scores’:{$elemMatch:{type :’exam’,score:{‘gt’:99.8}}}}).count();

Covered Queries

```
> for (i=0;i<1000000;i++) { db.numbers.insert({"i" :i,"j" : "j" + i,
"k":Math.floor(Math.random()*120),"created" : new Date()});}
```

- **db.numbers.ensureIndex({i:1,j:1,k:1})**

```
>var exp=db.numbers.find().explain(executionStats)
> exp.find({i:6,j:"j6",k:77})
{
    "queryPlanner" : {
        "plannerVersion" : 1,
"namespace" : "test.numbers",
"indexFilterSet" : false,
"parsedQuery" : {
        "$and" : [
                {
                        "i" : {
                                "$eq" : 6
                        }
                },
                {
                        "j" : {
                                "$eq" : "j6"
                        }
                },
                {
                        "k" : {
                                "$eq" : 77
                        }
                }
        ]
},
"winningPlan" : {
        "stage" : "FETCH",
        "inputStage" : {
                "stage" : "IXSCAN",
                "keyPattern" : {
                        "i" : 1,
                        "j" : 1,
                        "k" : 1
                },
                "indexName" : "i_1_j_1_k_1",
                "isMultiKey" : false,
                "isUnique" : false,
                "isSparse" : false,
                        "isPartial" : false,
                        "indexVersion" : 1,
                        "direction" : "forward",
                        "indexBounds" : {
                            "i" : [
                                "[6.0, 6.0]"
                            ],
                            "j" : [
                                "[\"j6\", \"j6\"]"
                            ],
                            "k" : [
                                "[77.0, 77.0]"
                            ]
                        }
                    }
                },
                "rejectedPlans" : [ ]
            },
            "executionStats" : {
                "executionSuccess" : true,
                "nReturned" : 1,
                "executionTimeMillis" : 4,
"totalKeysExamined" : 1,
"totalDocsExamined" : 1,   ( it scans the documents) ( this is not a covered query
becos _id is projected )
                "executionStages" : {
                    "stage" : "FETCH",
                    "nReturned" : 1,
                    "executionTimeMillisEstimate" : 0,
                    "works" : 2,
                    "advanced" : 1,
                    "needTime" : 0,
                    "needYield" : 0,
                    "saveState" : 0,
"restoreState" : 0,
"isEOF" : 1,
"invalidates" : 0,
"docsExamined" : 1,
"alreadyHasObj" : 0,
"inputStage" : {
        "stage" : "IXSCAN",
        "nReturned" : 1,
        "executionTimeMillisEstimate" : 0,
        "works" : 2,
        "advanced" : 1,
        "needTime" : 0,
        "needYield" : 0,
        "saveState" : 0,
        "restoreState" : 0,
        "isEOF" : 1,
        "invalidates" : 0,
        "keyPattern" : {
                "i" : 1,
                "j" : 1,
                "k" : 1
        },
        "indexName" : "i_1_j_1_k_1",
        "isMultiKey" : false,
        "isUnique" : false,
        "isSparse" : false,
        "isPartial" : false,
        "indexVersion" : 1,
        "direction" : "forward",
        "indexBounds" : {
                "i" : [
                        "[6.0, 6.0]"
                ],
                "j" : [
                                        "[\"j6\", \"j6\"]"
                                ],
                                "k" : [
                                        "[77.0, 77.0]"
                                ]
                        },
                        "keysExamined" : 1,
                        "dupsTested" : 0,
                        "dupsDropped" : 0,
                        "seenInvalidated" : 0
                }
        }
},
"serverInfo" : {
        "host" : "LIN73000604",
        "port" : 27017,
        "version" : "3.2.4",
        "gitVersion" : "e2ee9ffcf9f5a94fad76802e28cc978718bb7a30"
},
"ok" : 1
}
```

> exp.find({i:6,j:"j6",k:77},{_id:0,i:1,j:1,k:1}) (we need to project id:0 to make mongodb use only index scan which is called covered index)

```
{
        "queryPlanner" : {
                "plannerVersion" : 1,
                "namespace" : "test.numbers",
                "indexFilterSet" : false,
                "parsedQuery" : {
                        "$and" : [
                                {
                                        "i" : {
                                                "$eq" : 6
                                        }
```

```
				},
				{
					"j" : {
						"$eq" : "j6"
					}
				},
				{
					"k" : {
						"$eq" : 77
					}
				}
			]
		},
		"winningPlan" : {
			"stage" : "PROJECTION",
			"transformBy" : {
				"_id" : 0,
				"i" : 1,
				"j" : 1,
				"k" : 1
			},
			"inputStage" : {
				"stage" : "IXSCAN",
				"keyPattern" : {
					"i" : 1,
					"j" : 1,
					"k" : 1
				},
				"indexName" : "i_1_j_1_k_1",
				"isMultiKey" : false,
				"isUnique" : false,
				"isSparse" : false,
				"isPartial" : false,
				"indexVersion" : 1,
                                "direction" : "forward",
                                "indexBounds" : {
                                        "i" : [
                                                "[6.0, 6.0]"
                                        ],
                                        "j" : [
                                                "[\"j6\", \"j6\"]"
                                        ],
                                        "k" : [
                                                "[77.0, 77.0]"
                                        ]
                                }
                        }
                },
                "rejectedPlans" : [ ]
        },
        "executionStats" : {
                "executionSuccess" : true,
                "nReturned" : 1,
                "executionTimeMillis" : 7,
"totalKeysExamined" : 1,
"totalDocsExamined" : 0,  ( It scans only the index )
                "executionStages" : {
                        "stage" : "PROJECTION",
                        "nReturned" : 1,
                        "executionTimeMillisEstimate" : 0,
                        "works" : 2,
                        "advanced" : 1,
                        "needTime" : 0,
                        "needYield" : 0,
                        "saveState" : 0,
                        "restoreState" : 0,
                        "isEOF" : 1,
                        "invalidates" : 0,
"transformBy" : {
        "_id" : 0,
        "i" : 1,
        "j" : 1,
        "k" : 1
},
"inputStage" : {
        "stage" : "IXSCAN",
        "nReturned" : 1,
        "executionTimeMillisEstimate" : 0,
        "works" : 2,
        "advanced" : 1,
        "needTime" : 0,
        "needYield" : 0,
        "saveState" : 0,
        "restoreState" : 0,
        "isEOF" : 1,
        "invalidates" : 0,
        "keyPattern" : {
                "i" : 1,
                "j" : 1,
                "k" : 1
        },
        "indexName" : "i_1_j_1_k_1",
        "isMultiKey" : false,
        "isUnique" : false,
        "isSparse" : false,
        "isPartial" : false,
        "indexVersion" : 1,
        "direction" : "forward",
        "indexBounds" : {
                "i" : [
                        "[6.0, 6.0]"
                ],
                                "j" : [
                                        "[\"j6\", \"j6\"]"
                                ],
                                "k" : [
                                        "[77.0, 77.0]"
                                ]
                        },
                        "keysExamined" : 1,
                        "dupsTested" : 0,
                        "dupsDropped" : 0,
                        "seenInvalidated" : 0
                }
        }
},
"serverInfo" : {
        "host" : "LIN73000604",
        "port" : 27017,
        "version" : "3.2.4",
        "gitVersion" : "e2ee9ffcf9f5a94fad76802e28cc978718bb7a30"
},
"ok" : 1
}
```

Geospatial Indexes :
==================

Aloows us to find things based on location .

First we shall discuss about 2 dimensional model and then 3 d model .

```
C:\Users\srsenthi>mongo
2016-04-07T09:41:31.760+0530 I CONTROL  [main] Hotfix KB2731284 or later
update is not installed, will zero-out data files
MongoDB shell version: 3.2.4
```

```
connecting to: test
> db.stores.insert({"name":"rubys","type":"Barber","Location":[40,74]})
WriteResult({ "nInserted" : 1 })
> db.stores.insert({"name":"ACE
Hardware","type":"Hardware","Location":[40.232,-74.343]})
WriteResult({ "nInserted" : 1 })
> db.stores.insert({"name":"Tickel Candy","type":"food","Location":[41.232,-
75.343]})
WriteResult({ "nInserted" : 1 })

> db.stores.find()
{ "_id" : ObjectId("5705ef6912c458824b4eebb5"), "name" : "rubys", "type" :
"Barber", "Location" : [ 40, 74 ] }
{ "_id" : ObjectId("5705f08812c458824b4eebb6"), "name" : "ACE Hardware",
"type" : "Hardware", "Location" : [ 40.232, -74.343 ] }
{ "_id" : ObjectId("5705f0b012c458824b4eebb7"), "name" : "Tickel Candy",
"type" : "food", "Location" : [ 41.232, -75.343 ] }
> db.stores.find({Location:{$near:[66,74]}})

> db.stores.ensureIndex({Location:"2d",type:1})
{
	"createdCollectionAutomatically" : false,
	"numIndexesBefore" : 1,
	"numIndexesAfter" : 2,
	"ok" : 1
}
> db.stores.getIndexes()
[
	{
		"v" : 1,
		"key" : {
			"_id" : 1
		},
		"name" : "_id_",
		"ns" : "test.stores"
	},
	{
		"v" : 1,
		"key" : {
			"Location" : "2d",
			"type" : 1
		},
		"name" : "Location_2d_type_1",
		"ns" : "test.stores"
	}
]
> db.stores.find({Location:{$near:[50,50]}})
{ "_id" : ObjectId("5705ef6912c458824b4eebb5"), "name" : "rubys", "type" :
"Barber", "Location" : [ 40, 74 ] }
{ "_id" : ObjectId("5705f08812c458824b4eebb6"), "name" : "ACE Hardware",
"type" : "Hardware", "Location" : [ 40.232, -74.343 ] }
{ "_id" : ObjectId("5705f0b012c458824b4eebb7"), "name" : "Tickel Candy",
"type" : "food", "Location" : [ 41.232, -75.343 ] }
>
>db.places.insert({"name":"Apple Store","city":"Palo
alto","location":{"type":"Point","coordinates":[-
122.169129,37.4434854]},"type":"Retail"})

WriteResult({ "nInserted" : 1 })

> db.places.insert({"name":"Peninsula Ceremery","city":"Palo
alto","location":{"type":"Point","coordinates":[-
122.137044,37.423556]},"type":"Retail"})

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

```
> db.places.insert({"name":"Tamalpais State","city":"Mill
Valley","location":{"type":"Point","coordinates":[-
122.5995522,37.895943]},"type":"Park"})

WriteResult({ "nInserted" : 1 })

> db.places.insert({"name":"Fry's Electroincs","city":"Palo
alto","location":{"type":"Point","coordinates":[-
122.137044,37.423556]},"type":"Retail"})

WriteResult({ "nInserted" : 1 })


> db.places.ensureIndex({location:"2dsphere"})

{

        "createdCollectionAutomatically" : false,

        "numIndexesBefore" : 1,

        "numIndexesAfter" : 2,

        "ok" : 1

}

> db.places.getIndexes()

[

        {

                "v" : 1,

                "key" : {
```

```
                "_id" : 1
            },
            "name" : "_id_",
            "ns" : "test.places"
        },
        {
            "v" : 1,
            "key" : {
                "location" : "2dsphere"
            },
            "name" : "location_2dsphere",
            "ns" : "test.places",
            "2dsphereIndexVersion" : 3
        }
]

> db.places.find({location:{$near : {$geometry : { type : "Point",coordinates :[-
122.166641,37.4278925]},$maxDistance :2000}}}).pretty()
{
        "_id" : ObjectId("5706057e12c458824b4eebb8"),
        "name" : "Apple Store",
        "city" : "Palo alto",
        "location" : {
                "type" : "Point",
                "coordinates" : [
                        -122.169129,
                        37.4434854
                ]
        },
        "type" : "Retail"
}
```

Creating Index in the background:

The students collection already has index on student_id and scores.score array field .

If not available , create it as given below

```
> db.students.createIndex(['scores.score:1'])
```

This will take nearly 15 mins time , meanwhile , open another session and type the steps given below

In another terminal ,

- **Use school**
- **Db.students.findOne()**

  The request is still in process and the it blocks .

  It means that when we create the index in the foreground and try to do some querying on the same database from the other session , it blocks the reads .

  This can be avoided by creating the index in the background .

In first Terminal

```
> db.students.getIndexes()

[

        {

                "v" : 1,

                "key" : {

                        "_id" : 1

                },

                "ns" : "school.students",
                "name" : "_id_"
        },
        {
                "v" : 1,
                "key" : [
                        "scores.score:1"
                ],
                "ns" : "school.students",
                "name" : "0_scores.score:1"
        }
]
```

In second terminal,

- **Use school**
- **Db.students.findOne()**
- **..(response takes a lot of time … it blocks)**
- **Go back to terminal one and press ctl c ,it prompts if the index creation has to be killed .type yes**

Terminal One :

- **Create the index in the background ,**

```
> db.students.createIndex(['scores.score:1'],{background:true})
```

This would take some time , meanwhile do the seps given below in terminal two

```
 MongoDB shell version: 2.2.0
connecting to: test
> db.students.findOne()
null
> db.employees.findOne()
{
        "_id" : ObjectId("5700bf5c071d2db146924623"),
        "employee_id" : 1,
        "name" : "andrew",
        "cell" : 986055433
}
> use school
switched to db school
> db.students.findOne()
{
        "_id" : ObjectId("56e29884b5493e90ac7ec4d7"),
        "student_id" : 0,
        "scores" : [
                {
```

```
                "type" : "exam",
                "score" : 1.5629349419872596
            },
            {
                "type" : "quiz",
                "score" : 69.23197999806101
            },
            {
                "type" : "homework",
                "score" : 78.82490318401389
            },
            {
                "type" : "homework",
                "score" : 97.93998624112517
            }
        ],
        "class_id" : 386
}
```

What we find is that when we query the students database from the second session , it gives the results immediately without blocking unlike the first case … so when indexes are created in the background , still read operations are permitted in the second session

Index Examples

1 . Creating a 1 million documents

================================

```
for (i=0; i<1000000; i++) {
... db.users.insert(
... {
... "i" : i,

"username" : "user"+i,
... "age" : Math.floor(Math.random()*120),
... "created" : new Date()
... }
... );
...}
```

2. db.users.find({username: "user101"}).explain()

3. db.users.find({username: "user101"}).limit(1).explain()

4. db.users.ensureIndex({"username" : 1}) (Single Key Index)

5. db.users.find({"username" : "user101"}).explain()

6. db.users.find().sort({"age" : 1, "username" : 1})

7. db.users.ensureIndex({"age" : 1, "username" : 1}) (Compound Indexes)

8. db.users.find({}, {"_id" : 0, "i" : 0, "created" : 0})

9. db.users.find({"age" : 21}).sort({"username" : -1})

10. db.users.find({"age" : {"$gte" : 21, "$lte" : 30}}).sort({"username" :1}). hint({"username" : 1, "age" : 1}).

... explain()
11. db.users.ensureIndex({"username" : 1}, {"unique" : true}) (unique Index)
12. db.users.insert({username: "bob"})
E11000 duplicate key error index: test.users.$username_1 dup key: { : "bob" }

Profiling

=======

Creating Students collection

==========================

Loading 10 million documents in students collection in school database using the script given below

```
db=db.getSiblingDB("school");
db.students.drop();
```

```
types = ['exam', 'quiz', 'homework', 'homework'];
// 1 million students
for (i = 0; i < 1000000; i++) {

    // take 10 classes
    for (class_counter = 0; class_counter < 10; class_counter
++) {
      scores = []
          // and each class has 4 grades
          for (j = 0; j < 4; j++) {

      scores.push({'type':types[j],'score':Math.random()*100});
          }

      // there are 500 different classes that they can take
      class_id = Math.floor(Math.random()*501); // get a class
id between 0 and 500

      record = {'student_id':i, 'scores':scores,
'class_id':class_id};
      db.students.insert(record);

    }

}
```

```
Mongodb by default logs queries which take more 100ms in a log
file which is read at the time of start up
```

Profiler writes queries ,documents to system.profile

Three levels of profiler

Level 0 => profiling is off
level 1 => log slow queries
level 2 => log all queries ---general debugging feature – to see database traffic

Enable the profiler from the command line

Create a directory called proflog in d:\data\db

```
      mongod --dbpath D:\data\db\proflog --profile=1 --slowms=2
```

log in to mongo shell

Remove the index from students collection

```
db.students.find({student_id:10000})

db.system.profile.find().pretty()
> db.getProfilingLevel()
1
> db.getProfilingStatus()
{ "was" : 1, "slowms" : 2 }
> db.setProfilingLevel(1,4)
{ "was" : 1, "slowms" : 2, "ok" : 1 }
> db.setProfilingLevel(0)
{ "was" : 1, "slowms" : 4, "ok" : 1 }
> db.getProfilingStatus()
{ "was" : 0, "slowms" : 4 }
> db.getProfilingLevel()
0
```

Mongotop

Gives the info about time taken for reads and writes

Mongotop 3 (every 3 seconds it gives the details)

Mongostat → similar to iostat

Simulate the population of students collection with two storage engines configured on 27017 and 271018

Monitor the infor in both the ports using mongostat

From command line → 27017

- **Mongostat**

From 2nd CMD line -→ 27018

- **Mongostat –port 27018**