**DATABASE**

Table -> Collection

Row -> Document

Mongodb is case sensitive.

**Commands(We can use command prompt or terminal)**

**(1)To start mongo-db server:**

**mongosh**

**(2)To clear screen:**

**cls**

**(3)To show all Database:**

**show dbs; OR show databases;**

Note: it will not the particular database which has no collection/document.

**(4)To create new database:**

**use <database name>;**

Note: after creating database we need to hit all command inside database. Go to inside use below command:

use <database-name>

**(5)To delete database:**

**db.dropDatabase();**

**(6)To show all collections(tables):**

**show collections;**

**(7)To create new collection(table):**

**db.createCollection(‘<collection-name>’);**

Note: we need to hit command inside database.

**(8)To delete collection(table):**

**db.<collection-name>.drop();**

Note: we need to hit command inside database.

**(9)Inserting single document(row) in Collection(table):**

**db.<collection-name>.insertOne(**

**{ field1:value1, field2:value2 }**

**………….**

**);**

Note: we need to hit command inside database.

**Example:**

**db.students.insertOne(**

**{ “name”:”Raj”, number:9893098930, “roll no”:”22701” }**

**………….**

**);**

Note: we can use space b/w two strings that cases we need to use single or double quotes.

**(10)Inserting multiple documents(rows) in Collection(table):**

**db.<collection-name>.insertMany([**

**{ field1:value1, field2:value2 },**

**{ field1:value1, field2:value2 },**

**{ field1:value1, field2:value2 }**

**………….**

**]);**

Note:

1. if one of the document have some error then all documents(rows) will be insert before error document and after error will not insert any document by default.
2. If we want to insert all documents except error document, then we need to use below syntax:

db.<collection-name>.insertMany([

{ field1:value1, field2:value2 },

{ field1:value1, field2:value2 },

{ field1:value1, field2:value2 }

………….

],{ ordered:false });

**(11)Fetch all records(Documents):**

**db.<collection-name>.find()**

Note: It will show 21 records at a time, to show all records we need to use “it” command

**(12)Fetch all records with condition:**

**db.<collection-name>.find({<key>:<value>})**

**Ex: db.student.find({‘name’:’vinod’})**

**(13)Fetch 2 records:**

**db.<collection-name>.find().limit(2)**

**(14)Fetch 1 records:**

**db.<collection-name>.findOne({<key>:<value>})**

Note: it will fetch first record bydefault.

**(15) Importing JSON file into MongoDB:**

**mongoimport <jsonfile path> -d <database-name> -c <collection-name> //For without array format**

**mongoimport <jsonfile path> -d <database-name> -c <collection-name> --jsonArray //For array format**

Note: File size maximum 16mb.

**(16) Exporting data of MongoDB:**

**mongoexport -d <database-name> -c <collection-name> -o <path-where we want to save with file name.json>**

**Comparison Operator(to compare with numeric value)**

**Equal to($eq), Not equal to($ne), Greater than($gt), Greater than equal to($gte),**

**Less than($lt), Less than equal to($lte), Multiple eual to - In($in), Multiple not eual to - Not in($nin)**

**To fetch records with comparison operator**

**(17) Equal to ($eq)**

**db.<collection-name>.find({ ‘field-name’ : { $eq : <value> } })**

Ex: db.products.find({ ‘price’ : { $eq : 799 } })

Note: To find no. of total documents(rows), use below syntax:

db.products.find({ ‘price’ : { $eq : 799 } }).count()

**(18) Not Equal to ($ne)**

**db.<collection-name>.find({ ‘field-name’ : { $ne : <value> } })**

Ex: db.products.find({ ‘price’ : { $ne : 799 } })

Note: To find no. of total documents(rows), use below syntax:

db.products.find({ ‘price’ : { $ne : 799 } }).count()

Note:

Similaraly we use Greater than($gt), Greater than equal to($gte), Less than($lt), Less than equal to($lte)

**(19) in ($in)**

**db.<collection-name>.find({ ‘field-name’ : { $in : [<value1>,<value2>,<value3>,<value4>] } })**

Ex: db.products.find({‘price’ : { $in : [ 78,45,34,23 ] } })

Note: it will fetch all records whre price is equal to 78,45,34 and 23

**(20) nin ($nin)**

**db.<collection-name>.find({ ‘field-name’ : { $nin : [<value1>,<value2>,<value3>,<value4>] } })**

Ex: db.products.find({‘price’ : { $nin : [ 78,45,34,23 ] } })

Note: it will fetch all records whre price is not equal to 78,45,34 and 23

**Cursor Methods**

**count(), limit(), skip(), sort()**

Note: we need to use these methods in ending of syntax

**(21) count()**

**db.<collection-name>.find({ ‘field-name’ : { $eq : 788 } }).count()**

Ex: db.products.find({ ‘price’ : { $eq : 799 } }).count()

Note: it will show total no. of records.

**(22) limit()**

**db.<collection-name>.find({ ‘field-name’ : { $eq : 788 } }).limit(<no. of row>)**

Ex: db.products.find({ ‘price’ : { $eq : 799 } }).limit(1)

Note: it will show only 1 record (first record). For fetching 3 records we need write limit(3)

**(23) skip()**

**db.<collection-name>.find({ ‘field-name’ : { $eq : 788 } }).limit(<no. of row>).skip()**

Ex: db.products.find({ ‘price’ : { $eq : 799 } }).limit(3).skip(1)

Note: it will select only 3 record (first 3 record) , skip 1 record( first record ) and show 2 records

**(24) sort()**

**db.<collection-name>.find({ ‘field-name’ : { $eq : 788 } }).limit().sort({ ‘<field>’ : <1 or -1> })**

Ex: db.products.find({ ‘price’ : { $eq : 799 } }).limit(3).sort({ ‘price’ : 1 })

Note: it will show only 3 record (first 3 record) by accending order. For descending order put -1

**Logical Operators**

**Logical and($and), logical OR($or), Logical Not($not) , Logical Nor($nor),**

**(25) Logical and($and)**

**db.<collection-name>.find({ $and :[ {<condition1>},{<condition2>} ]})**

Ex: db.products.find({ $and : [ { ‘price’ : { $gt : 100 } },{ ‘name’ : ‘Diamind Ring’ } ] })

OR

db.products.find({ ‘price’ : { $gt : 100 } , ‘name’ : ‘Diamond Ring’ })

Note: All conditions must be true.

**(26) Logical or($or)**

**db.<collection-name>.find({ $or :[ {<condition1>},{<condition2>} ]})**

Ex: db.products.find({ $or : [ { ‘price’ : { $gt : 100 } },{ ‘name’ : ‘Diamind Ring’ } ] })

Note: Atleast one condtion must be true from all of the condtions.

**(27) Logical not($not)**

**db.<collection-name>.find({ ‘<field-name>’ : { $not : { <condition> } } })**

Ex: db.products.find({ ‘price’ : { $not : { $eq : 100 } } })

Note: one condition must not be true.

**(28) Logical nor($nor)**

**db.<collection-name>.find({ $nor :[ {<condition1>},{<condition2>} ]})**

Ex: db.products.find({ $nor : [ { ‘price’ : { $gt : 100 } },{ ‘name’ : ‘Diamind Ring’ } ] })

Note: All conditions must not be true.

**Complex Expression [ignore]**

**(29) Expression($expr)**

**db.<collection-name>.find ({ $expr : { <operator> : [ ‘$<field-name>’ , <value> ] } })**

Ex: db.products.find({ $expr : { $gt : [ ‘$price’ , 1340 ] } })

Note: It will show all records where price is greater 1340

Ex: db.sales.find({ $expr : { $gt : [ { $multiply : [ ‘$quantity’ , ’$price’ ] } , ‘$totprice’ ] } })

Note:

1. multiply, price and totprice all are colum names. It will show those records where multiplication of quantity and price colum value must be greater than totprice colum. (i.e. (quantity \* price) > totprice)
2. we can also use $add or $sub instead of $multiply for addition and subtraction .

**Elements Operator**

**$exists , $type , $size**

**(30) $exists**

**db.<collection-name>.find ({ <field-name> : { $exists : <boolean> } })**

Ex: db.products.find({ price : { $exists : true } })

Note: it will show all records where price colum is exist. For not exist we need to use false.

Ex: db.products.find({ price : { $exists : true }, price : { $gt : 1250 } })

Note: it will show all records where price colum is exist and value of this colum is greater than 1250

**(31) $type**

**db.<collection-name>.find ({ <field-name> : { $type : “<bson-data-type>” } })**

Ex: db.products.find({ price : { $type : ‘number’ } })

Note: It will show all records where price colum’s data type is number. We can also use string to check. For Boolean we need to use bool

**(32) $size**

**db.<collection-name>.find ({ <field-name> : { $size : <array-length> } })**

Ex: db.products.find({ ‘comments’ : { $size : 1 } })

Note: Normaly in colum field data has simple format link: ‘name’ : ‘Raj’ but comments is a field which has data in json format with array index, that means it is called nested data(nested documents) in json format.

Like:

[ {

\_id : 8,

title : ‘this is title’ ,

comments [

{‘name’ : ’rajkumar’ , ’rollno’ : ’567’ , ’comments’ : ’no’},

{‘name’ : ’raj’ , ’rollno’ : ’567’ , ’comments’ : [{‘sub’:’eng’},{‘sts’:’ok’}]},

]

}]

Here in first row comment colum has no json data and in second row comments colum has 2 records in json format.

i.e. it will show all records where comments colum has one record.

**Projection**

**(33) To fetch records for a particular colum**

**db.<collection-name>.find ({ <field-name> : { $size : <array-length> } }, <field-name> : <value> )**

Ex: db.products.find({ ‘comments’ : { $size : 1 } }, comments : 1 , author : 1 , \_id : 0)

Note: it will show only comments and author colums data. Use 0 for not showing and use 1 for showing. We need to use 1,1,1…. Or 0,0,0,……. We can’t use 0,1,1,0,0,1,0,1

**Embeded Documents(Nested Documents)**

**(34)**

**db.<collection-name>.find ({ ‘<parent-field-name>.<child-field-name>’ , ‘<value>’ })**

suppose we have a below record:

[ {

\_id : 8,

title : ‘this is title’ ,

metadata: { views : 2500 , likes : 4000 }

comments [

{‘name’ : ’rajkumar’ , ’rollno’ : ’567’ , ’comments’ : ’no’},

{‘name’ : ’raj’ , ’rollno’ : ’567’ , ’comments’ : [{‘sub’:’eng’},{‘sts’:’ok’}]},

]

}]

Ex: db.comments.find({ ‘comments.name’ , ‘raj’ })

Note: it will show all comments field data where raj is the value of name field.

**(35)**

**db.<collection-name>.find ({ ‘<parent-field-name>.<child-field-name>’ , <operator> : <value> })**

suppose we have a below record:

[ {

\_id : 8,

title : ‘this is title’ ,

metadata: { views : 2500 , likes : 4000 }

comments [

{‘user : ‘Henry’ , ’rollno’ : ’560’ , ’comments’ : ’no’},

{‘user’ : ’rajkumar’ , ’rollno’ : ’567’ , ’comments’ : ’no’},

{‘user’ : ’raj’ , ’rollno’ : ’567’ , ’comments’ : [{‘sub’:’eng’},{‘sts’:’ok’}]},

]

}]

Ex: db.comments.find({ ‘metadata.views’ , $gt : 1200 })

Note: it will show all records views is greater than 1200 inside metadata colum.

Ex: db.comments.find({ ‘comments.user’ : ‘Henry’ , ‘metadata.likes’ : { $gt : 50 } })

Note: it will show all records where Henry is the value of user colom inside comments colum and likes is greater than 50 inside metadata colum.

**$all and $elemMatch**

**( Both are valid only for embeded(nested) documents )**

$all : it is used for single child colum

$elemMatch : it is used for double or multiple child colum

**(36)**

**db.<collection-name>.find ({ ‘<parent-field-name>.<child-field-name>’ , { $all : [‘<value1>’,’<value2>’] } })**

suppose we have a below record:

[ {

\_id : 8,

title : ‘this is title’ ,

metadata: { views : 2500 , likes : 4000 }

comments [

{‘user : ‘Alice’ , ’rollno’ : ’560’ , ’comments’ : ’no’},

{‘user’ : ’Vinod’ , ’rollno’ : ’567’ , ’comments’ : ’no’},

{‘user’ : ’raj’ , ’rollno’ : ’567’ , ’comments’ : [{‘sub’:’eng’},{‘sts’:’ok’}]},

]

}]

Ex: db.comments.find({ ‘comments.user’ : { $all : [‘Alice’,’Vinod’] } })

Note: it will show all records where the value of user colum is Alice and Vinod inside comments colum

**(37)**

**db.<collection-name>.find ({ ‘<parent-field-name>’ : { $elemMatch : { ‘<child-field-name>’ : ‘<value>’ , ‘<child-field-name>’ : ‘<value>’ } } })**

suppose we have a below record:

[ {

\_id : 8,

title : ‘this is title’ ,

metadata: { views : 2500 , likes : 4000 }

comments [

{‘user’ : ‘Alice’ , ’rollno’ : ’560’ , ’comments’ : ’no’},

{‘user’ : ’Vinod’ , ’rollno’ : ’567’ , ’comments’ : ’no’},

{‘user’ : ’raj’ , ’rollno’ : ’567’ , ’comments’ : [{‘sub’:’eng’},{‘sts’:’ok’}]},

]

}]

Ex: db.comments.find({ ‘comments’ : { $elemMatch : { ‘user’ : ’Vinod’ , ‘rollno’ : ‘560’ } } })

Note: it will show all records where the value of user colum is Vinod and value of rollno colum is 560 inside comments colum.

**Updation Operation**

**(38) Update One Record:**

**db.<collection-name>.updateOne ({ <colum-name> : ‘<value>’} , { $set : { ‘<colum-name>’ : ‘<value>’ } })**

Ex: db.comments.updateOne({ \_id : ObjectId(“56hucttcf5”) } , { $set : { ‘price’ : 45 } } )

Ex: db.comments.updateOne({ name : ‘ravi’ } , { $set : { ‘status’ : true } } )

Note: it will update the value of price field and status field.

**(38) Update Many Record:**

**db.<collection-name>.updateMany ({ <colum-name> : ‘<value>’} , { $set : { ‘<colum-name>’ : ‘<value>’ } })**

Ex: db.comments.updateMany({ price : 120 } , { $set : { ‘status’ : true } } )

Ex: db.comments.updateMany({ name : ‘ravi’ } , { $set : { ‘status’ : true } } )

Note: it will update the value of status field where price is 120 and name is ravi.

**Renaming and Removing colum name**

**(39) Renaming column name for one record:**

**db.<collection-name>.updateOne ({ <colum-name> : ‘<value>’} , { $rename : { ‘<existing-colum-name>’ : ‘<new-colum-name>’ } })**

Ex: db.comments.updateOne({ price : 120 } , { $rename : { ‘status’ : ‘prodctstatus’ } } )

Note: it will update the colum name status with productstatus for one records.

**(40) Renaming column name for many record:**

**db.<collection-name>.updateMany ({ <colum-name> : ‘<value>’} , { $rename : { ‘<existing-colum-name>’ : ‘<new-colum-name>’ } })**

Ex: db.comments.updateMany({ price : 120 } , { $rename : { ‘status’ : ‘prodctstatus’ } } )

Note: it will update the colum name status with productstatus for multiple records.

**(41) Delete one column for one record:**

**db.<collection-name>.updateOne ({ <colum-name> : ‘<value>’} , { $unset : { ‘<existing-colum-name>’ : 1 } })**

Ex: db.comments.updateOne({ \_id : 5 } , { $unset : { ‘status’ : 1 } } )

Note: it will delete the status colum where id=5 for one record.

**(42) Delete one column for many records:**

**db.<collection-name>.updateMany ({ <colum-name> : ‘<value>’} , { $unset : { ‘<existing-colum-name>’ : 1 } })**

Ex: db.comments.updateOne({ price : 50 } , { $unset : { ‘status’ : 1 } } )

Note: it will delete the status colum where id=5 for one record.

**Updating Arrays and Embeded(Nested) Documents**

**(43) Adding new colum for one record:**

**db.<collection-name>.updateOne ({ <colum-name> : ‘<value>’} , { $set : { ‘<new-colum-name>’ : ‘<value>’ } })**

Ex: db.comments.updateOne({ \_id : 5 } , { $set: {‘user’ : ‘jessy’}} )

Note: it will add one new coum user for one document where id =5

**(44) Adding one document(row) in nested documents:**

suppose we have a below record:

[ {

\_id : 8,

title : ‘this is title’ ,

metadata: { views : 2500 , likes : 4000 }

comments [

{‘user : ‘Alice’ , ’rollno’ : ’560’ , ’comments’ : ’no’},

{‘user’ : ’Vinod’ , ’rollno’ : ’567’ , ’comments’ : ’no’},

{‘user’ : ’raj’ , ’rollno’ : ’567’ , ’comments’ : [{‘sub’:’eng’},{‘sts’:’ok’}]},

]

}]

**db.<collection-name>.updateOne ({ <colum-name> : ‘<value>’} , { $push : { ‘<parent-colum-name>’ : { <child-coumn-name> : ‘<value>’ , <child-colum-name> : <value> } } })**

Ex: db.comments.updateOne({ \_id : 5 } , { $push: {comments : { user : ‘Eva’ , rollno : 1000 } }} )

Note: it will add one new record inside comments field which is nested documents where id=5

**db.<collection-name>.updateOne ({ <colum-name> : ‘<value>’} , { $pop : { ‘<parent-colum-name>’ : 1 } })**

Ex: db.comments.updateOne({ \_id : 5 } , { $pop: { ‘comments’ : 1}} )

Note: it will remove one last record inside comments field which is nested documents where id=5

**db.<collection-name>.updateOne ({ <colum-name> : ‘<value>’ , ‘<parent-colum-name>.<child-colum-name>’ : ‘<value of child colum>’ } , { $set : { ‘<parent-colum-name>.$.<child-colum-name>’ : ‘<put value>’} })**

Ex: db.comments.updateOne({ \_id : 5 , ‘comments.user’ : ‘Alice’} , { $set: { ‘comments.$.rollno’ : ‘Awesome’}} )

Note: it will update the value of rollno, where the value of user is Alice inside comment field(nested documents) where id=5

**Delete Operations**

**(45) Delete records in collection:**

**db.<collection-name>.deleteOne({ <colom-name> : <value> })**

Ex: db.comments.deleteOne({ \_id : 5 })

Note: it will delete one document(row) inside collection(table)

Ex: db.comments.deleteMany({ price : 55 })

Note: it will delete all rows where price = 55

**Indexes**

(it performs faster query in comparison to normal query)

**Benefit:**

1. faster query
2. efficient sorting
3. improve aggregation
4. indexing on multiple fields
5. we use index to reduce execution time.
6. If our work will done by execution 100 records than we don’t require to execute 10000 records, on that case we can use indexes.

**When not use index:**

1. Where filter is not require to particular colom or any thing else.

**Note: we can work without using indexes.**

**(46) we can find all execution details by explain() method:**

Ex: db.comments.deleteMany({ price : 55 }).explain()

Note: it will show all execution details.

Ex: db.comments.deleteMany({ price : 55 }).explain(‘executionStats’)

Note: it will show all execution details in deeply.

**(47) To get indexes:**

**db.<collection-name>.getIndexes()**

Ex: db.comments.getIndexes()

Note: it will show all indexes.

**(48) To create index:**

**db.<collection-name>.createIndex(‘<field-name>’ : < put 0 or 1>)** // 1 for Ascending , 0 for Descending order

Ex: db.comments.createIndex(‘name’ : 1)

Note : it will generate one index suppose: name\_1

Ex: db.comments.explain(‘executionStats’).find({ ‘name’ : ‘Air Fryer’ })

Note: it will show execution details in deeply, filter name colum and we can check execution time , it has reduced.

**(49) To delete index:**

**db.<collection-name>.dropIndex(‘<field-name>’ : < put 0 or 1>)** // 1 for Ascending , 0 for Descending order

Ex: db.comments.dropIndex(name : 1)

Note: it will delete the created index suppose: name\_1

Ex: db.comments.createIndex({ email : 1 } , { unique : true })

Note: it will create a index for email field that is unique. We can not insert duplicate email.

Ex: db.comments.createIndex(field : “text”)

Ex: db.comments.find({ $text : { $search : “keyword” } }) //to filter

**Aggregation**

**( $match , $group, $sort, $project, $push, $unwind, $addToset,$size,$limit,$skip,$filter)**

( We can perform complex query chaining process )

**(50) :**

**db.<collection-name>.aggregate([{ $match : { ‘<colum-name>’ : ‘<value>’ } }])**

Ex: db.comments.aggregate([{ $match : { ‘name’ : ‘sleek wooden’ } }])

Note: it will show all records where name = ‘sleek wooden’ inside collection.

Ex: db.comments.aggregate([{ $match : { price : { $gt : 1250 } } }])

Note: it will show all records where price > 1250

Ex:

db.comments.aggregate([{

$group : {

\_id : ‘$company’ ,

totalProducts : { $sum : 1 }

}

}])

Note: company is a field name, we need to use $ symbol by self, totoalProducts is a self defined colum , it is not exist in collection. $sum is also self defined. It will show result group by company colum.

Ex:

db.comments.aggregate([{

$match : {

company : ‘09juhyg56fsttsg’

}

}])

Ex:

db.comments.aggregate([{

$group : {

\_id : ‘$company’ ,

totalProducts : { $sum : ‘$price’ }

}

}])

Note: it will show sum of price colum data where id=company. company and price existing colum. totalProducts is self defined colum

Ex:

db.comments.aggregate([

{

$match : { price : { $gt : 900 } }

},

{

$group : {

\_id : ‘$company’ ,

totalProducts : { $sum : ‘$price’ }

}

}

])

Note: it will find price which is greater than 900 than find sum(i.e. chaining process). price is existing colum. totalProducts is self defined colums.

Ex:

db.comments.aggregate([

{

$match : { quantity : 5 }

},

{

$group : {

\_id : ‘$quantity’ ,

totalPrice : { $sum : ‘$price’ },

totalAvg : { $avg : ‘$price’ }

}

}

])

Note: it will find quantity which is equal to 5 than find total and avg of price. totalPrice and totalAvg both are self defined colums. price is existing colum.

Ex:

db.comments.aggregate([

{

$match : { price : { $gt : 900 } }

},

{

$group : {

\_id : ‘$category’ ,

totalPrice : { $sum : ‘$price’ }

}

},

{

$sort : { totalPrice : 1 }

}

])

Note: it will find price which is greater than 900, than it will perform sort[ asceding(1) or descending(-1) order ] than find sum(i.e. chaining process). price is existing colum. totalPrice is self defined colums.

Ex:

db.comments.aggregate([

{

$project : {

price : 1

}

}

])

Note: it will show all price colum data with id colum.

Ex:

db.comments.aggregate([

{

$project : {

price : 1 , \_id : 0

}

}

])

Note: it will show only price colum data.

Ex:

db.comments.aggregate([

{

$project : {

price : 1 ,

discountPrice : { $multiply : [ ‘$price’ , 0.8 ] }

}

}

])

Note: it will show price colum (existing colum) with actual data and discountPrice(self defined colum) with 0.8% of actual data.

Ex:

db.comments.aggregate([

{

$match : { price : { $gt : 900 } }

},

{

$project : {

price : 1 ,

discountPrice : { $multiply : [ ‘$price’ , 0.8 ] }

}

}

])

Note: it will find price colum data which is greater than 900 than show price colum (existing colum) with actual data and discountPrice(self defined colum) with 0.8% of actual data. It will perform chaining process.

Ex:

db.comments.aggregate([

{

$unwind : ‘$colors’ //to destructure of colors colum data

},

{

$match : { price : { $gt : 900 } } // to find price colum data which is > 900

},

{

$group : {

\_id : ‘$price’ ,

allColors : { $push : ‘$colors’ } // to push colors colum data inside allColors colum

}

}

])

Note:

1. To remove duplicate data we need to use $addToSet instead of $push
2. It will show colors colum data(data will merge/group) where price > 900 with name of allColors coum,
3. colors is existing colum and allColors, colorLength both are self defined colum.

Ex:

db.comments.aggregate([

{

$match : { price : { $gt : 900 } } // to find price colum data which is > 900

},

{

$unwind : ‘$colors’ //to destructure of colors colum data

},

{

$group : {

\_id : ‘$price’ ,

allColors : { $push : ‘$colors’ } // to push colors colum data inside allColors colum

}

},

{

$project : {

\_id : 1,

colors : 1,

colorLength : { $size : ‘$colors’ }

}

},

{

$limit : 1 // to show 1 record, if don’t required remove this part

}

])

Note:

1. To remove duplicate data we need to use $addToSet instead of $push
2. It will show colors colum data(data will merge/group) where price > 900 with name of allColors coum,
3. colors is existing colum and allColors, colorLength both are self defined colum.
4. We can’t use $size operator inside $group
5. It will follow chaining process.
6. We can use $skip instead of $limit. It will skip one record.

Ex:

db.comments.aggregate([{

$project : { name : 1,

thapaValue : {

$filter : {

input : ‘$values’,

as : ‘val’ ,

cond : { $gt : [‘$$val’ , 30] }

}

}

}

}])

Note:

1. cond, $gt, $filter, $project input, cond all are predefined .
2. name , values are existing colums.
3. val is self defined colum for values colum
4. it will show all data which is greater than 30 name of val

**MongoDB Atlas,**

**MongoDB Driver,**

**Connectivity with MongoDB Local and Atlas with CRUD(Node js)**

**(4:15)**