1. Differences between the SQL and NO-SQL Database.
2. MongoDb Introduction.

2.1) what is MongoDB?

2.2) Advantages of MongoDB.

2.3) Where is MongoDB used?

3. Installation of MongoDb.

3.1) Installation.

3.2)MongoDB Shell

3.3)MongoDB Server.

3.4)MongoDB Driver.

3.5) Display the version of Shell and Server.

3.6) MongoDB physical DB and MongoDB Logical DB.

1. Database.

4.1) Default Databases.

4.2) Creating Database.

4.3) Droping Database.

5. Collection.

5.1)Creating Collection.

5.2)Droping Collection.

6. Document.

6.1) Create Document.

6.1.1)Insert document into collection from .js file.

6.1.2)Insert document into collection from .json and .csv file.

6.1.3)Nested Document.

6.1.4)Arrays in Document.

6.1.5) Object Id.

6.1.6)Ordered property.

6.1.7)WriteConcern property.

6.1.8)Atomicity.

6.2) Read Document.

6.2.1)find

6.2.2)findOne

6.2.3)Selection Criteria with find and findOne

6.2.4)Querying the nested Document.

6.2.5)Querying the Array elements in Document.

6.2.6)Quering Operators

6.2.6.1) Comparision Operator.

6.2.6.2) Logical Operator.

6.2.6.3) Element Query Operators

a)$exists b)$type.

6.2.6.4) Query Evaluation Operator.

a)$expr b)$regex c)$mod d)$jsonSchema e)$text f)$where.

6.2.6.5)Array query operators.

a)$all b)$eleMatch c)$size

6.3) Update Document.

6.3.1)Update operators.

6.3.2)upsert command.

6.3.3)Array Update operators.

6.3.4)Adding Element to Array in Document.

6.3.5)$addToSet

6.3.6)Removing the Elements using $pop operator from Array.

6.4) Delete Document.

7. Cursor

8. Projection.

9. MongoDb utilities(Database Tools)

9.1)mongoimport.

9.2)mongoexport.

9.3)mongodump.

9.4)mongorestore.

10. GUI Tools for mongoDB operations.

11. Indexing.

12. Aggeragate Frame Works.

12.1)Aggregate Pipeline

1. Difference between SQL and NO- SQL Databases.

`

i) The sql db are primarly called as RDBMS or Relataional DBs.

ii)Sql db has fixed schema or static schema or predefined schema.

EX:- Data is stored in tables. The table has fixed schema.

EMP(eno,ename,sal);

Here The record can only contain eno, ename and sal. It can’t contain the other information. Because of it has fixed schema.

iii) SQL db are not suitable for hierarchical data storage.

iv)SQL db is best suited for complex quieries.

v)Example to sql databases are :- oracle,Sybase,mysql,…etc.

B)NO-SQL DB:- (No-Sql Db stands for Not Only sql Database).

i) The N0-SQL db are called as Non-Relational DB’s.

ii) No-SQL db has dynamic schema.

The document can contain required employee information.

EX:-

Document1:-{ ‘name’: ‘suku’,

‘Eno’:7

}

Document2:-

{ ‘name’: ‘veena’;

‘age’:39;

}

The documents has different schemas. The documents can’ be placed in mongodb.

iii) No-Sql db is suitable for hierarchical data storage.

iv) No-Sql db is not suited to complex queiries.

v) Example to no-sql databases are MongoDB,Casandra,H-base..etc.

2.MongoDb Introduction:- The MongoDb is no-sql db. It is document based and non-relational Database. MongoDB was developed by a NewYork based organization named 10gen which is now known as MongoDB Inc.

MongoDb open source is available aswell as paid software is available for commercial organizations.

Advantages:-

a) MangoDb is schema less.Therefore we can store un structure data like videos,images ,..etc.

b) There are no complex joins in mongodb.

c) we can scale Mongodb horizontally. This scalling is also very easy.

d) It uses internal memory for storing working sets and this is the reason of its fast access.

e) It is 100 times faster than traditional database systems.

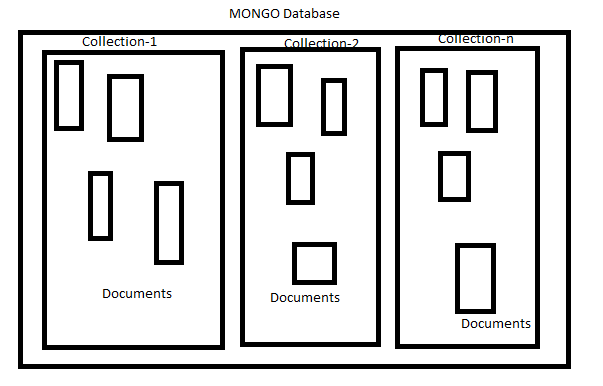
f) The document contains the data in json format. There fore any programming language can understand the data with out conversions Where is mongodb used?

The mongo db is used to handle/manage the huge data. It is used in desktop applications,mobile applications and web applications.

2.1)MongoDB Structure:- MongoDB Physical database contains several logical databases.

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

Each collection contains several documents. Document is something like record/row in relational database. The document is independent to each other.



eg:

Database: Shopping cart database

Collections: Customers, Products, Orders

Cusomer Collection: contains several documents

document1:

{

"Name":"Sunny",

"age":40,

"Salary":10000

}

document2:

{

"Name":"Durga"

}

document-3:

{

"name":"Bunny",

"age":30,

"address":

{

"city":"Hyderabad"

},

5

"hobbies":[

{"name":"Cricket playing"},

{"name":"swimming"}

]

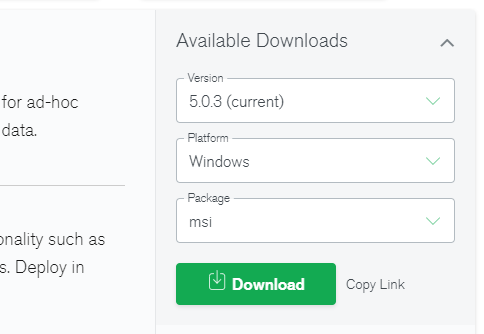
}

3)MongoDb Installation:-

3.1)Installation:-

Step1: Go to following URL <http://www.mongodb.org/downloads>**.**

****Step2: The website automatically knows the current system operating system name . It displays the current mongodb version which is compatible with current os.****

********

****Step3. Click download button. After clicking the download button , MongoDb Executable file will be downloaded into local machine.****

****Step4: double click the Mongodb executable file.****

****Step5. Click the next button in setup wizard.****

****Step6. Select check box to agree the licence.****

****Step7. Select custom or complete based on your requirement.****

****Step8. Enter location where Mongodb will be installed.****

****While installing the MangoDB , The two applications are installed automatically. The two components are****

1. ****MongoDB Server.****
2. ****MongoDB shell.****

****3.2) MongoDB Server:- It is java script based application. The server takes commands from MongoDB shell and executes them.****

****Totally, The Mongodb is handled and managed by MongoDB server. Management means createing, deleting, updating …etc.****

****The mongo DB can be in local machine or remote machine.****

****It is started/launched with ‘mongod’ command.****

****There are two types of Servers.****

****a)community version:- It is used for education purpose.****

****b)Enterprise version :- It has more features regarding to security,roles,..etc than community version. It is used in real time environment.****

****3.3) MongoDB Shell:- It is also java script based application. It is interface for MongoDb administrator and Mongodb Developer to test queryies and operations with DB.****

****MongoDB shell is available in two forms.****

****1.CLI(Commond line interface)****

****2 GUI(Graphical User interface):- Compass,Robo T3 are example to GUI’s.****

****Main responsibility is collecting the Queiries/commonds from user and giving them to Mongo db server. Collecting queries result from mongodb server and displaying them to user.****

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

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

****EX:- PyMongo module is example to MangoDb Driver. It is used by python.****

****3.5) Display version of Shell and Server:-****

****In My System , I Installed Mongodb in following location.****

****Location:- C:\\**** ****\Program Files\MongoDB****

****Step1. Open the command prompt.****

****Step2. Go to bin directory in MongoDB Directory structure.****

****C;\\**** ****\Program Files\MongoDB\Server\5.0\bin****

****Stp3. Run the following command to display shell version.****

* ****Mongo -version****

****C:\Program Files\MongoDB\Server\5.0\bin>mongo -version****

****MongoDB shell version v5.0.3****

****Build Info: {****

****"version": "5.0.3",****

****"gitVersion": "657fea5a61a74d7a79df7aff8e4bcf0bc742b748",****

****"modules": [],****

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

****"environment": {****

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

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

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

****}****

****}****

****Step4:- To display server version use following command.****

****Mongod –version.****

****C:\Program Files\MongoDB\Server\5.0\bin>mongod -version****

****db version v5.0.3****

****Build Info: {****

****"version": "5.0.3",****

****"gitVersion": "657fea5a61a74d7a79df7aff8e4bcf0bc742b748",****

****"modules": [],****

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

****"environment": {****

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

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

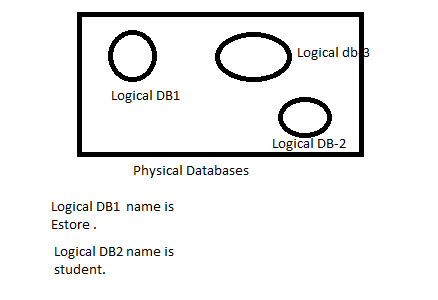
****"target\_arch": "x86\_64"****

****}****

****}****

****3.6)MongoDB physical DB and MongoDB Logical DB:-****

****When I install mongodb, the physical db is automatically created. In that User created db is called logical db.****

********

4.Database:-

4.1) Default Database:- when we installed MongoDB, The physical DB has 3 default logical databases. They are

a) admin 0.000GB

b)config 0.000GB

c)local 0.000GB

1. Admin:- admin database is used to store user authentication and authorization information like usernames,passwords,roles etc.This database is used by administrators while creating,deleting and updating users and while assigning roles.
2. Config:- To store configuration information of mongodb server.
3. Local:- local database can be used by admin while performing replication process.

EX:- show dbs

admin 0.000GB

config 0.000GB

local 0.000GB

4.2)Creating DB:- MongoDb especially did not provide any command to create a database.

Syntax:- use Dbname

If there is no Database, then new database will be created and control will be switched to newly created db.

Note:- immediately after creating the DB, It will not be added to database list.

Whenever we create first collection then , This database is added to Database list.

If there is database with given name, then it will return the existing database.

Example:-

-------------

use College

switched to db College

> show dbs

admin 0.000GB

config 0.000GB

local 0.000GB

4.3) Dropping DB:- The following command deletes only current database and its all associated files.

Syntax:- db.dropDatabase();

Note:- Where db is implicit pointer which points to the current database.

Example:-

> db.dropDatabase()

{ "ok" : 1 }

4.4) display the current database name:-

Syntax: db

Example:-

> db

College

5.Collections:-

5.1)Creating Collections:- The following syntax is used to create Collection.

Syntax:- db.createCollection(‘name’,options);

Options:- It is optional parameter. It is document type. It specifies memory size and index on collection.

a)max:- It specifies the maximum number of documents allowed in the capped collection.

b)size:- It specifies a maximum size in bytes for a capped collection. Ifcapped is true, then you need to specify this field also.

c)AutoIndexId:- If it is set to true, automatically create index on ID field. Its default value is false.

d)Capped:- If it is set to true, enables a capped collection. Capped collection is a fixed size collecction that automatically overwrites its oldest entries when it reaches its maximum size. If you specify true, you need to specify size parameter also.

Example:-1

> db.createCollection('student');

{ "ok" : 1 }

> show dbs

College 0.000GB

admin 0.000GB

config 0.000GB

local 0.000GB

Example:2

db.createCollection('staff',{capped:true,max:1,size:100})

{ "ok" : 1 }

Note:- MongoDB creates collections automatically when you insert some documents. Even collection does not exists,it creates collection first and then it create document in that collection.

Syntax:- db.collectionname.insert({})

Example:-

> db.principal.insert({name:'sukumar'})

WriteResult({ "nInserted" : 1 })

> show dbs

College 0.000GB

admin 0.000GB

config 0.000GB

local 0.000GB

> show collections

principal

staff

student

5.2)Dropping Collections:- The following syntax is used to drop the collection.

Syntax:- db.collectionname.drop()

Example:-

> show collections

principal

staff

student

> db.principal.drop()

true

> show collections

staff

student

6.Document:- Usually, The document contains the json format data.

6.1)Create Document:- The insert command inserts new document in collection. .

Syntax:- db.Collectionname.insert(document);

Example:-1

> db.staff.insert({name:'suku',age:39})

WriteResult({ "nInserted" : 1 })

The new document has been inserted in staff collection.

Example:-2

> db.student.insert([{name:'sula',age:14},{name:'suma',age:12},{name:'suha',age:14}]);

BulkWriteResult({

"writeErrors" : [ ],

"writeConcernErrors" : [ ],

"nInserted" : 3,

"nUpserted" : 0,

"nMatched" : 0,

"nModified" : 0,

"nRemoved" : 0,

"upserted" : [ ]

})

> db.student.find();

{ "\_id" : ObjectId("61755271278d76fe7b787d1f"), "name" : "sv", "age" : 36 }

{ "\_id" : ObjectId("617559a2278d76fe7b787d22"), "name" : "veena", "age" : 40 }

{ "\_id" : ObjectId("61755afc278d76fe7b787d23"), "name" : "sula", "age" : 14 }

{ "\_id" : ObjectId("61755afc278d76fe7b787d24"), "name" : "suma", "age" : 12 }

{ "\_id" : ObjectId("61755afc278d76fe7b787d25"), "name" : "suha", "age" : 14 }

Example:3

-------------

var one={name:'sulamaha',age:41};

> db.student.insert(one);

WriteResult({ "nInserted" : 1 })

> db.student.find();

{ "\_id" : ObjectId("61755271278d76fe7b787d1f"), "name" : "sv", "age" : 36 }

{ "\_id" : ObjectId("617559a2278d76fe7b787d22"), "name" : "veena", "age" : 40 }

{ "\_id" : ObjectId("61755afc278d76fe7b787d23"), "name" : "sula", "age" : 14 }

{ "\_id" : ObjectId("61755afc278d76fe7b787d24"), "name" : "suma", "age" : 12 }

{ "\_id" : ObjectId("61755afc278d76fe7b787d25"), "name" : "suha", "age" : 14 }

{ "\_id" : ObjectId("61755b50278d76fe7b787d26"), "name" : "sulamaha", "age" : 41 }

Note:- we can place duplicate copies of documents in same collection. But their object ids are different.

6.1.1)Insert document into collection from .js file:-

Step1. Create a .js file. Place the insert commands in .js file.

Step2. The Load command will gets insertion commands from .js file. Then after, these commands are executed at mongo shell. When these commands are executed, the documents are placed in specified collection.

Example:

Abc.js

---------

db.student.insert({'name':'rock','age':57});

db.student.insert({'name':'hhh','age':65});

-----------------------------------------------------------------------------------------------------------------------

> db.student.find();

{ "\_id" : ObjectId("61755271278d76fe7b787d1f"), "name" : "sv", "age" : 36 }

{ "\_id" : ObjectId("617559a2278d76fe7b787d22"), "name" : "veena", "age" : 40 }

{ "\_id" : ObjectId("61755afc278d76fe7b787d23"), "name" : "sula", "age" : 14 }

{ "\_id" : ObjectId("61755afc278d76fe7b787d24"), "name" : "suma", "age" : 12 }

{ "\_id" : ObjectId("61755afc278d76fe7b787d25"), "name" : "suha", "age" : 14 }

{ "\_id" : ObjectId("61755b50278d76fe7b787d26"), "name" : "sulamaha", "age" : 41 }

{ "\_id" : ObjectId("61755b95278d76fe7b787d27"), "name" : "sulamaha", "age" : 41 }

> load('E:\abc.js');

true

> db.student.find();

{ "\_id" : ObjectId("61755271278d76fe7b787d1f"), "name" : "sv", "age" : 36 }

{ "\_id" : ObjectId("617559a2278d76fe7b787d22"), "name" : "veena", "age" : 40 }

{ "\_id" : ObjectId("61755afc278d76fe7b787d23"), "name" : "sula", "age" : 14 }

{ "\_id" : ObjectId("61755afc278d76fe7b787d24"), "name" : "suma", "age" : 12 }

{ "\_id" : ObjectId("61755afc278d76fe7b787d25"), "name" : "suha", "age" : 14 }

{ "\_id" : ObjectId("61755b50278d76fe7b787d26"), "name" : "sulamaha", "age" : 41 }

{ "\_id" : ObjectId("61755b95278d76fe7b787d27"), "name" : "sulamaha", "age" : 41 }

{ "\_id" : ObjectId("617564a1278d76fe7b787d2a"), "name" : "rock", "age" : 57 }

{ "\_id" : ObjectId("617564a1278d76fe7b787d2b"), "name" : "hhh", "age" : 65 }

6.1.2)

A)**Insert document into collection from .json file**:- To insert document from .json file,we should need ‘mongoimport’ tool. By default it is not available.

Step1. [Download MongoDB Command Line Database Tools | MongoDB](https://www.mongodb.com/try/download/database-tools).

Click th Tools.

Click the download button.

Step2.Download mongoimport.exe file. Place it in bin folder of Mongodb software.

Step3. mongoimport command should be executed from the command prompt but not from the shell.

Syntax:- mongoimport --db databaseName --collection collectionName --file

fileName --jsonArray

Note:- 1.mongoimport creates database and collection automatically if not available.

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

B**) Insert Documents into collection from .csv file**:-

Syntax:- mongoimport –db databaseName -c collectioname --type CSV --headerline --maintainInsertionOrder --ignoreBlanks --drop filename.

Note:- 1# --headerline is used to ensure that our csv headerline should be read as keys while importing into database.

2. # --maintainInsertionOrder ensures that the data will be inserted in the top to bottom order from the csv file (row by row).It is an optional.

3. --ignoreBlanks ---> Whereever the value for key is blank inside csv file, that particular key itself will not be imported to database for that particular document. It is also an optional.

Example:

-----------

C:\Program Files\MongoDB\Server\5.0\bin>mongoimport --db Library -c mca --type CSV --headerline --drop D:/Book1.csv

2021-11-03T10:17:55.165+0530 connected to: mongodb://localhost/

2021-11-03T10:17:55.167+0530 dropping: Library.mca

2021-11-03T10:17:55.479+0530 4 document(s) imported successfully. 0 document(s) failed to import.

> use Library

switched to db Library

> db.Mca.find()

> db.mca.find()

{ "\_id" : ObjectId("6182147baaa527fd94d17fd1"), "SNO" : 1, "SNAME" : "x", "Marks" : 77 }

{ "\_id" : ObjectId("6182147baaa527fd94d17fd2"), "SNO" : 4, "SNAME" : "abc", "Marks" : 80 }

{ "\_id" : ObjectId("6182147baaa527fd94d17fd3"), "SNO" : 2, "SNAME" : "y", "Marks" : 78 }

{ "\_id" : ObjectId("6182147baaa527fd94d17fd4"), "SNO" : 3, "SNAME" : "z", "Marks" : 79 }

6.1.3) Nested Document:- Sometimes we can take a document inside another document, such type of documents are called nested documents or embedded documents.

Ex:-

Book={ ‘title’:’DBMS’,’ isbn’:’123’,’ downloadable’:true,

‘Address’:{ street:’kamati street’, dist:’nlr’, dr-no:’367’},’publisher’:’TataMCGrawhill’,

‘Author’:{‘name’:’korth’,’exp’:3,’city’:’ny’}

}

6.1.4) Arrays In Document:- The single key may have one or more than one value. The collection of values is said to be array. The values may be string,number or objects type.

EX:-

Book={‘title’:’DBMS’,

‘author’:’korth’,

‘languages’:[‘telugu’,’english’,’tamil’]

}

6.1.5) Object-Id:-

The MongoDb server assign a unique Id to every document. This is nothing but unique-Id.

It is something like primary key in relational db. This id will be assigned to ‘\_Id ‘field of collection.

We can’t modify the ‘\_id’ value once we Inserted the document in the collection.

The object-id is not json type. It is BSON type. It consists of 12 bytes.

1. The first 4 bytes represents the timestamp when this document was inserted.

2. The next 3 bytes represents machine identifier( host name)

3. The next 2 bytes represents process id.

4. The last 3 bytes represents some random increment value.

Examples:-

--------------

> db.student.find();

{ "\_id" : ObjectId("61755271278d76fe7b787d1f"), "name" : "sv", "age" : 36 }

{ "\_id" : ObjectId("617559a2278d76fe7b787d22"), "name" : "veena", "age" : 40 }

{ "\_id" : ObjectId("61755afc278d76fe7b787d23"), "name" : "sula", "age" : 14 }

{ "\_id" : ObjectId("61755afc278d76fe7b787d24"), "name" : "suma", "age" : 12 }

{ "\_id" : ObjectId("61755afc278d76fe7b787d25"), "name" : "suha", "age" : 14 }

{ "\_id" : ObjectId("61755b50278d76fe7b787d26"), "name" : "sulamaha", "age" : 41 }

{ "\_id" : ObjectId("61755b95278d76fe7b787d27"), "name" : "sulamaha", "age" : 41 }

{ "\_id" : ObjectId("617564a1278d76fe7b787d2a"), "name" : "rock", "age" : 57 }

{ "\_id" : ObjectId("617564a1278d76fe7b787d2b"), "name" : "hhh", "age" : 65 }

> db.student.find()[0]

{ "\_id" : ObjectId("61755271278d76fe7b787d1f"), "name" : "sv", "age" : 36 }

> db.student.find()[1]

{

"\_id" : ObjectId("617559a2278d76fe7b787d22"),

"name" : "veena",

"age" : 40

}

> db.student.find().\_id;

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

**ObjectId("61755271278d76fe7b787d1f")**

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

**ISODate("2021-10-24T12:32:49Z")**

Note:- 1 By using \_id field, we can provide our own value as ObjectId. MongoDB server will generate default ObjectId iff we are not providing any \_id field value.

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

2. If we provide same id for two document, then we get error.

Example:-

-----------

**db.student.insert({"\_id":77,"name":"sv",'age':34});**

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

**> db.student.insert({"\_id":77,"name":"sv1",'age':134});**

**WriteResult({**

**"nInserted" : 0,**

**"writeError" : {**

**"code" : 11000,**

**"errmsg" : "E11000 duplicate key error collection: College.student index: \_id\_ dup key: { \_id: 77.0 }"**

**}**

**})**

**6.1.6)** Ordered Bulk Insertion:- we can insert the bulk documents using insert or insertMany command.

Syntax: db.collectioname.insert([{},{}{},…]);

Db.collectionname.insertMany([{},{}{},…]);

The Default behaviour of bulk insertion/insertion Many command is

* While performing the bulk operation, if there is no errors in any documents then all documents will be placed in collection.

Suppose, if any document insertion fails, then rest of the documents will be not inserted and inserted document can’t be rollbacked.

Example:

> db.student.insert([{'\_id':2,'name':'suku','age':40},{'\_id':2,'name':'rock', age:39},{'\_id':3,'name':'sv','age':41}]);

BulkWriteResult({

"writeErrors" : [

{

"index" : 1,

"code" : 11000,

"errmsg" : "E11000 duplicate key error collection: College.student index: \_id\_ dup key: { \_id: 2.0 }",

"op" : {

"\_id" : 2,

"name" : "rock",

"age" : 39

}

}

],

"writeConcernErrors" : [ ],

"nInserted" : 1,

"nUpserted" : 0,

"nMatched" : 0,

"nModified" : 0,

"nRemoved" : 0,

"upserted" : [ ]

})

> db.student.find()

{ "\_id" : 2, "name" : "suku", "age" : 40 }

>

The default functionality of bulk insert/insertMany commands can be changed using ‘ordered’ property.

Syntax: insert/insertMany([{},{},..],{ordered:false});

If ordered is true then those command has default behaviour.

If ordered is false then those command behaviour is customized. If any document insertion fails then MongoDB leaves that document and inserts rest of the documents.

Example:

-----------

db.student.insert([{'\_id':2,'name':'suku','age':40},{'\_id':2,'name':'rock', age:39},{'\_id':3,'name':'sv','age':41}],{ordered:false});

BulkWriteResult({

"writeErrors" : [

{

"index" : 1,

"code" : 11000,

"errmsg" : "E11000 duplicate key error collection: College.student index: \_id\_ dup key: { \_id: 2.0 }",

"op" : {

"\_id" : 2,

"name" : "rock",

"age" : 39

}

}

],

"writeConcernErrors" : [ ],

"nInserted" : 2,

"nUpserted" : 0,

"nMatched" : 0,

"nModified" : 0,

"nRemoved" : 0,

"upserted" : [ ]

})

> db.student.find();

{ "\_id" : 2, "name" : "suku", "age" : 40 }

{ "\_id" : 3, "name" : "sv", "age" : 41 }

6.1.7)writeConcern Proeprty:-

USECASE:1 Whenever we are performing insert operation, bydefault the shell/client will wait until getting acknowledgement. Server will provide acknowledgement after completing insert operation. This may reduce performance at client side.

> db.cars.insertOne({I:"Innova"})

{

"acknowledged" : true,

"insertedId" : ObjectId("5fe800f68a9854ae87538e1b")

}

We can customize this behaviour using ‘writeConcern’ property.

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

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

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

Example:-

------------

> db.student.insert({'name':'suha','age':09},{writeConcern:{w:0}})

WriteResult({ })

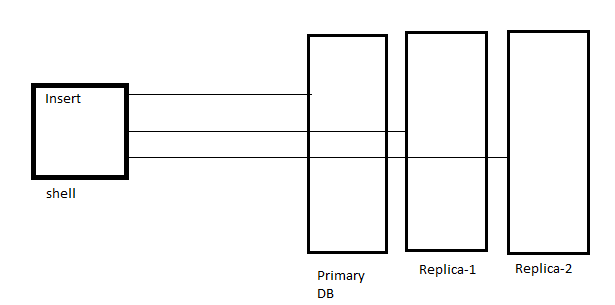
Note:- If lakhs of records are required to insert, if one or two document insertion fails still no problem, but performance is important then writeConcern is the best choice.

UseCase2:- In Production , for every database we have to maintain cloned/replica database because

1. To handle Fail over situations

2. For Load Balancing Purposes

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



After inserting how many instance, you are expecting acknowdgement, we can specify this by using writeConcern propery.

if w: 0 ===> No acknowledgement.

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

49

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

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

6.1.8)Atomicity:-

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

Ans: Whatever fields already added will be rollbacked.

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

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

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

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

6.2) Retrieve Document:-

6.2.1)find():- The find command read all documents from collection.

Syntax1:-db.collectionname.find();

Example:-

> db

College

> show collections

student

> db.student.insert({name:'suku',age:39});

WriteResult({ "nInserted" : 1 })

> db.student.insert({name:'sv',age:36});

WriteResult({ "nInserted" : 1 })

> db.student.find();

{ "\_id" : ObjectId("6175525a278d76fe7b787d1e"), "name" : "suku", "age" : 39 }

{ "\_id" : ObjectId("61755271278d76fe7b787d1f"), "name" : "sv", "age" : 36 }

The find command got all documents from the student collection.

To display documents in understandable way, we use pretty() method.

**Syntax:- db.collection name.find().pretty();**

Example:-

> db.student.find().pretty();

{

"\_id" : ObjectId("6175525a278d76fe7b787d1e"),

"name" : "suku",

"age" : 39

}

{ "\_id" : ObjectId("61755271278d76fe7b787d1f"), "name" : "sv", "age" : 36 }

6.2.2) findOne:- It read the only first document from the collection.

Syntax:-1 db.collectioname.findOne()

Example:

-----------

> db.student.findOne();

{ "\_id" : 2, "name" : "suku", "age" : 40 }

> db.student.find();

{ "\_id" : 2, "name" : "suku", "age" : 40 }

{ "\_id" : 3, "name" : "sv", "age" : 41 }

{ "\_id" : ObjectId("617f80a1943737c5a2913f19"), "name" : "suma", "age" : 12 }

{ "\_id" : ObjectId("617f80fa943737c5a2913f1a"), "name" : "suma", "age" : 12 }

{ "\_id" : ObjectId("617f8173943737c5a2913f1b"), "name" : "suha", "age" : 9 }

>

**Note:-1 pretty() and count() methods can be used on find() result but not on findOne() result.**

**6.2.3)**Selection Criteria with Find and findOne commands:- we can select the required document from collection using criteria.

Syntax:- db.collectionname.find({query});

This command will select the documents from collection . All selected documents defenitly met the specified condition in query.

Example:

-----------

> db.student.find();

{ "\_id" : 2, "name" : "suku", "age" : 40 }

{ "\_id" : 3, "name" : "sv", "age" : 41 }

{ "\_id" : ObjectId("617f80a1943737c5a2913f19"), "name" : "suma", "age" : 12 }

{ "\_id" : ObjectId("617f80fa943737c5a2913f1a"), "name" : "suma", "age" : 12 }

{ "\_id" : ObjectId("617f8173943737c5a2913f1b"), "name" : "suha", "age" : 9 }

> db.student.find({'name':'suma'});

{ "\_id" : ObjectId("617f80a1943737c5a2913f19"), "name" : "suma", "age" : 12 }

{ "\_id" : ObjectId("617f80fa943737c5a2913f1a"), "name" : "suma", "age" : 12 }

**Syntax:- db.collectionname.findOne({query});**

This command will select only one and first document which met the specified condition in query.

Examples:-

> db.student.find();

{ "\_id" : 2, "name" : "suku", "age" : 40 }

{ "\_id" : 3, "name" : "sv", "age" : 41 }

{ "\_id" : ObjectId("617f80a1943737c5a2913f19"), "name" : "suma", "age" : 12 }

{ "\_id" : ObjectId("617f80fa943737c5a2913f1a"), "name" : "suma", "age" : 12 }

{ "\_id" : ObjectId("617f8173943737c5a2913f1b"), "name" : "suha", "age" : 9 }

> db.student.findOne({'name':'suma'});

{

"\_id" : ObjectId("617f80a1943737c5a2913f19"),

"name" : "suma",

"age" : 12

}

6.2.4)Querying the Nested Document:- If the value of a field /key is again a document, then that nested property value can be accessed by using dot operator. In this case, key must be enclosed within quotes.

Example:

-----------

> db.student.find()

> db.student.insert({

... 'name':'suku',

... 'age':39,

... address:{ 'lmark':'chakalistreet',

... 'street':'rajaka stree',

... 'dist':'nlr'

... }

... });

WriteResult({ "nInserted" : 1 })

> db.student.insert({ 'name':'suku', 'age':39, address:{ 'lmark':'chakalistreet1', 'street':'rajaka street1', 'dist':'nlr' } });

WriteResult({ "nInserted" : 1 })

> db.student.find({"address.lmark":"chakalistreet"});

{ "\_id" : ObjectId("617f8ed8943737c5a2913f1c"), "name" : "suku", "age" : 39, "address" : { "lmark" : "chakalistreet", "street" : "rajaka stree", "dist" : "nlr" } }

6.2.5)Querying the Array elements in Document:-

Syntax:1 db.collectionname.find({‘keyname’:’value’});

Syntax:2 db.collectionname.find({‘keyname’:[‘value’]});

IfThe key should have array as its value and that array should has only one value that to specified value then only that document will be selected.

Syntax:3 db.collectinname.find({‘keyname’:[‘v1’,’v2’,…’vn’]});

Where v1,v2,.. vn order is an important. If array contains all values in specified order then only the document is selected.

Example:

-----------

db.student.insert({'name':'suku', 'language':['tel','eng','hin']});

WriteResult({ "nInserted" : 1 })

> db.student.find({'language':'tel'});

{ "\_id" : ObjectId("617f92a6943737c5a2913f1e"), "name" : "suku", "language" : [ "tel", "eng", "hin" ] }

> db.student.find({'language':['hin']});

> db.student.find({'language':['tel','eng','hin']});

{ "\_id" : ObjectId("617f92a6943737c5a2913f1e"), "name" : "suku", "language" : [ "tel", "eng", "hin" ] }

> db.student.find({'language':['eng','hin','tel']});

> db.student.find({'language':['tel','eng']});

6.2.6)Querying Operators:-

6.2.6.1)Comparision Opertors:- The comparision operators are

A) $lt b)$lte c)$gt d)$gte e) $eq f) $ne g) $in h)$nin

Syntax:-- {keyname:{ Operatorname:value}}

Examples:

-------------

> db.student.find({name:{$eq:'suku'}});

{ "\_id" : ObjectId("617f92a6943737c5a2913f1e"), "name" : "suku", "language" : [ "tel", "eng", "hin" ] }

> db.student.find({age:{$gt:25}});

> db.student.find({age:{$gt:15}});

{ "\_id" : ObjectId("617fb61d024f1cbf78a7bc7f"), "name" : "sv", "age" : 20 }

{ "\_id" : ObjectId("617fb630024f1cbf78a7bc80"), "name" : "sv1", "age" : 25 }

{ "\_id" : ObjectId("617fb63c024f1cbf78a7bc81"), "name" : "suma", "age" : 25 }

6.2.6.2)Logical Operators:- The logical operators are

a)$or b)$nor c)$and d)$not

syntax:- {Logical-operator:[{‘keyname’:{c-operator:value}},……]}

Examples:-

> db.student.find();

{ "\_id" : ObjectId("617f92a6943737c5a2913f1e"), "name" : "suku", "language" : [ "tel", "eng", "hin" ] }

{ "\_id" : ObjectId("617fb61d024f1cbf78a7bc7f"), "name" : "sv", "age" : 20 }

{ "\_id" : ObjectId("617fb630024f1cbf78a7bc80"), "name" : "sv1", "age" : 25 }

{ "\_id" : ObjectId("617fb63c024f1cbf78a7bc81"), "name" : "suma", "age" : 25 }

{ "\_id" : ObjectId("617fb645024f1cbf78a7bc82"), "name" : "suha", "age" : 15 }

> db.student.find({$or:[{'age':{$eq:20}},{'age':{$eq:15}}]});

{ "\_id" : ObjectId("617fb61d024f1cbf78a7bc7f"), "name" : "sv", "age" : 20 }

{ "\_id" : ObjectId("617fb645024f1cbf78a7bc82"), "name" : "suha", "age" : 15 }

> db.student.find({$and:[{'age':{$gt:20}},{'name':{$eq:'suma'}}]});

{ "\_id" : ObjectId("617fb63c024f1cbf78a7bc81"), "name" : "suma", "age" : 25 }

> db.student.find({$nor:[{'age':{$gt:20}},{'name':{$eq:'suma'}}]});

{ "\_id" : ObjectId("617f92a6943737c5a2913f1e"), "name" : "suku", "language" : [ "tel", "eng", "hin" ] }

{ "\_id" : ObjectId("617fb61d024f1cbf78a7bc7f"), "name" : "sv", "age" : 20 }

{ "\_id" : ObjectId("617fb645024f1cbf78a7bc82"), "name" : "suha", "age" : 15 }

Shortcut for AND operation:

--------------------------

MongoDB provides an implicit AND operation when specifying a

comma separated list of expressions.

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

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

Limitation of this shortcut:

----------------------------

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

Q. List out all students whose marks are >=50 and <= 90?

> db.students.find({marks: {$gte: 50}, marks: {$lte: 90}}).pretty()

{ "\_id" : ObjectId("5fed4de50df1d8f23c0f678c"), "name" : "B", "marks" : 20 }

{ "\_id" : ObjectId("5fed4de50df1d8f23c0f678d"), "name" : "I", "marks" : 90 }

{ "\_id" : ObjectId("5fed4de50df1d8f23c0f678e"), "name" : "C", "marks" : 30 }

{ "\_id" : ObjectId("5fed4de50df1d8f23c0f678f"), "name" : "A", "marks" : 10 }

{ "\_id" : ObjectId("5fed4de50df1d8f23c0f6791"), "name" : "E", "marks" : 50 }

{ "\_id" : ObjectId("5fed4de50df1d8f23c0f6792"), "name" : "G", "marks" : 70 }

{ "\_id" : ObjectId("5fed4de50df1d8f23c0f6793"), "name" : "D", "marks" : 40 }

{ "\_id" : ObjectId("5fed4de50df1d8f23c0f6794"), "name" : "F", "marks" : 60 }

{ "\_id" : ObjectId("5fed4de50df1d8f23c0f6795"), "name" : "H", "marks" :80 }

Reason: In Javascript object, duplicate keys are not allowed. If we are

trying to add duplicate keys then old value will be replaced with new

value.

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

It will become72

{marks: {$lte: 90}}

Solution: we should use $and operator

6.2.6.3) Element Query Operators:- MongoDb provided two element Query operators.

a)$exists b)$type.

a)$exists:-

syntax:- {fieldname:{$exists: true|false}}

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

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

Example:-

-----------

> db.student.find()

{ "\_id" : ObjectId("617f92a6943737c5a2913f1e"), "name" : "suku", "language" : [ "tel", "eng", "hin" ] }

{ "\_id" : ObjectId("617fb61d024f1cbf78a7bc7f"), "name" : "sv", "age" : 20 }

{ "\_id" : ObjectId("617fb630024f1cbf78a7bc80"), "name" : "sv1", "age" : 25 }

{ "\_id" : ObjectId("617fb63c024f1cbf78a7bc81"), "name" : "suma", "age" : 25 }

{ "\_id" : ObjectId("617fb645024f1cbf78a7bc82"), "name" : "suha", "age" : 15 }

> db.student.find({language:{$exists:true}});

{ "\_id" : ObjectId("617f92a6943737c5a2913f1e"), "name" : "suku", "language" : [ "tel", "eng", "hin" ] }

> db.student.find({language:{$exists:false}});

{ "\_id" : ObjectId("617fb61d024f1cbf78a7bc7f"), "name" : "sv", "age" : 20 }

{ "\_id" : ObjectId("617fb630024f1cbf78a7bc80"), "name" : "sv1", "age" : 25 }

{ "\_id" : ObjectId("617fb63c024f1cbf78a7bc81"), "name" : "suma", "age" : 25 }

{ "\_id" : ObjectId("617fb645024f1cbf78a7bc82"), "name" : "suha", "age" : 15 }

$type:- $type operator selects the documents where the value of the field is of a particular type.

Syntax:1 {fieldname:{$type:<Bson Type>}} It is for querying the single type.

Syntax:2 {fieldname:{$type:[<Bson Type>,<Bson Type>,<Bson Type>…]}} This is for querying the multiple types.

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

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

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

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

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

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

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

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

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

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

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

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

76

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

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

Q1. What is the difference between int and long?

-------------------------------------------------

int --->32 bits integer value

long --->64 bits integer value

Q2. What is the difference between double and decimal?

-------------------------------------------------

double --->64 bits floating point value

decimal --->128 bits floating point value

Note:

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

int

long

double

decimal

Examples:-

> db.student.find()

{ "\_id" : ObjectId("617f92a6943737c5a2913f1e"), "name" : "suku", "language" : [ "tel", "eng", "hin" ] }

{ "\_id" : ObjectId("617fb61d024f1cbf78a7bc7f"), "name" : "sv", "age" : 20 }

{ "\_id" : ObjectId("617fb630024f1cbf78a7bc80"), "name" : "sv1", "age" : 25 }

{ "\_id" : ObjectId("617fb63c024f1cbf78a7bc81"), "name" : "suma", "age" : 25 }

{ "\_id" : ObjectId("617fb645024f1cbf78a7bc82"), "name" : "suha", "age" : 15 }

> db.student.find({name:{$type:'string'}});

{ "\_id" : ObjectId("617f92a6943737c5a2913f1e"), "name" : "suku", "language" : [ "tel", "eng", "hin" ] }

{ "\_id" : ObjectId("617fb61d024f1cbf78a7bc7f"), "name" : "sv", "age" : 20 }

{ "\_id" : ObjectId("617fb630024f1cbf78a7bc80"), "name" : "sv1", "age" : 25 }

{ "\_id" : ObjectId("617fb63c024f1cbf78a7bc81"), "name" : "suma", "age" : 25 }

{ "\_id" : ObjectId("617fb645024f1cbf78a7bc82"), "name" : "suha", "age" : 15 }

> db.student.find({name:{$type:'int'}});

> db.student.find({age:{$type:'age'}});

Error: error: {

"ok" : 0,

"errmsg" : "Unknown type name alias: age",

"code" : 2,

"codeName" : "BadValue"

}

> db.student.find({age:{$type:'int'}});

> db.student.find({age:{$type:'number'}});

{ "\_id" : ObjectId("617fb61d024f1cbf78a7bc7f"), "name" : "sv", "age" : 20 }

{ "\_id" : ObjectId("617fb630024f1cbf78a7bc80"), "name" : "sv1", "age" : 25 }

{ "\_id" : ObjectId("617fb63c024f1cbf78a7bc81"), "name" : "suma", "age" : 25 }

{ "\_id" : ObjectId("617fb645024f1cbf78a7bc82"), "name" : "suha", "age" : 15 }

> db.student.find({language:{$type:'number'}});

> db.student.find({language:{$type:'string'}});

{ "\_id" : ObjectId("617f92a6943737c5a2913f1e"), "name" : "suku", "language" : [ "tel", "eng", "hin" ] }

> db.student.insert({"name":"rock","pno":["9703393965",9703393967,NumberLong("980328765")]});

WriteResult({ "nInserted" : 1 })

> db.student.find({pno:{$type:["string","long"]}});

{ "\_id" : ObjectId("6181eabca5580a8ff851c56b"), "name" : "rock", "pno" : [ "9703393965", 9703393967, NumberLong(980328765) ] }

6.2.6.4) Query Evaluation Operators:- We have following query Evaluation operators.

a)$expr b)$regex c)$mod d)jsonSchema e)$text f)$where

a)$expr:- expr means expression.Evaluate expression and select documents which satisfy that expression.

Syntax:

{ $expr: {<expression>}}

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

Examples:-

-------------

> db.student.find()

{ "\_id" : ObjectId("6181ee25a5580a8ff851c56c"), "name" : "suku", "age" : 39, "exp" : 25 }

{ "\_id" : ObjectId("6181ee31a5580a8ff851c56d"), "name" : "suku1", "age" : 39, "exp" : 125 }

{ "\_id" : ObjectId("6181ee49a5580a8ff851c56e"), "name" : "suku1", "age" : 25, "exp" : 25 }

> db.student.find({$expr:{$gt:["$age","$exp"]}});

{ "\_id" : ObjectId("6181ee25a5580a8ff851c56c"), "name" : "suku", "age" : 39, "exp" : 25 }

> db.student.find({$expr:{$eq:["$age","$exp"]}});

{ "\_id" : ObjectId("6181ee49a5580a8ff851c56e"), "name" : "suku1", "age" : 25, "exp" : 25 }

> db.student.find({$expr:{$ne:["$age","$exp"]}});

{ "\_id" : ObjectId("6181ee25a5580a8ff851c56c"), "name" : "suku", "age" : 39, "exp" : 25 }

{ "\_id" : ObjectId("6181ee31a5580a8ff851c56d"), "name" : "suku1", "age" : 39, "exp" : 125 }

b) regex: regex means regular expression.

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

Syntax:

-------

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

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

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

{ field: { $regex: /pattern/<options>}}

{ field: /pattern/<options>}

Where

i)pattern:-

EX:- / narayana / :- If field value contain word ‘Narayana’ then that document is selected.

/^e/:- if field value starts with letter ‘e’ then that document is selected.

/^[ef]/:- If field value starts with letter ‘e’ or ‘f’ then that document is selected.

/e$/:- If field value ends with letter ‘e’ then that document is selected.

/[ef]$/:- if field value ends with letter ‘e’ of ‘f’ then that document is selected.

ii)<options>:- $options: I (i) means ignore case.

Examples:-

> db.student.find();

{ "\_id" : ObjectId("6181ee25a5580a8ff851c56c"), "name" : "suku", "age" : 39, "exp" : 25 }

{ "\_id" : ObjectId("6181ee31a5580a8ff851c56d"), "name" : "suku1", "age" : 39, "exp" : 125 }

{ "\_id" : ObjectId("6181ee49a5580a8ff851c56e"), "name" : "suku1", "age" : 25, "exp" : 25 }

{ "\_id" : ObjectId("6181f282a5580a8ff851c56f"), "name" : "asuku1", "age" : 25, "exp" : 25 }

{ "\_id" : ObjectId("6181f288a5580a8ff851c570"), "name" : "asuku2", "age" : 25, "exp" : 25 }

> db.student.find({name:{$regex:/suku/}});

{ "\_id" : ObjectId("6181ee25a5580a8ff851c56c"), "name" : "suku", "age" : 39, "exp" : 25 }

{ "\_id" : ObjectId("6181ee31a5580a8ff851c56d"), "name" : "suku1", "age" : 39, "exp" : 125 }

{ "\_id" : ObjectId("6181ee49a5580a8ff851c56e"), "name" : "suku1", "age" : 25, "exp" : 25 }

{ "\_id" : ObjectId("6181f282a5580a8ff851c56f"), "name" : "asuku1", "age" : 25, "exp" : 25 }

{ "\_id" : ObjectId("6181f288a5580a8ff851c570"), "name" : "asuku2", "age" : 25, "exp" : 25 }

> db.student.find({name:{$regex:/^a/}});

{ "\_id" : ObjectId("6181f282a5580a8ff851c56f"), "name" : "asuku1", "age" : 25, "exp" : 25 }

{ "\_id" : ObjectId("6181f288a5580a8ff851c570"), "name" : "asuku2", "age" : 25, "exp" : 25 }

> db.student.find({name:{$regex:/^[as]/}});

{ "\_id" : ObjectId("6181ee25a5580a8ff851c56c"), "name" : "suku", "age" : 39, "exp" : 25 }

{ "\_id" : ObjectId("6181ee31a5580a8ff851c56d"), "name" : "suku1", "age" : 39, "exp" : 125 }

{ "\_id" : ObjectId("6181ee49a5580a8ff851c56e"), "name" : "suku1", "age" : 25, "exp" : 25 }

{ "\_id" : ObjectId("6181f282a5580a8ff851c56f"), "name" : "asuku1", "age" : 25, "exp" : 25 }

{ "\_id" : ObjectId("6181f288a5580a8ff851c570"), "name" : "asuku2", "age" : 25, "exp" : 25 }

> db.student.find({name:{$regex:/1$/}});

{ "\_id" : ObjectId("6181ee31a5580a8ff851c56d"), "name" : "suku1", "age" : 39, "exp" : 125 }

{ "\_id" : ObjectId("6181ee49a5580a8ff851c56e"), "name" : "suku1", "age" : 25, "exp" : 25 }

{ "\_id" : ObjectId("6181f282a5580a8ff851c56f"), "name" : "asuku1", "age" : 25, "exp" : 25 }

> db.student.find({name:{$regex:/[12]$/}});

{ "\_id" : ObjectId("6181ee31a5580a8ff851c56d"), "name" : "suku1", "age" : 39, "exp" : 125 }

{ "\_id" : ObjectId("6181ee49a5580a8ff851c56e"), "name" : "suku1", "age" : 25, "exp" : 25 }

{ "\_id" : ObjectId("6181f282a5580a8ff851c56f"), "name" : "asuku1", "age" : 25, "exp" : 25 }

{ "\_id" : ObjectId("6181f288a5580a8ff851c570"), "name" : "asuku2", "age" : 25, "exp" : 25 }

> db.student.find({name:{$regex:/[12]$/,$options:i}});

uncaught exception: ReferenceError: i is not defined :

@(shell):1:39

> db.student.find({name:{$regex:/[12]$/,$options:'i'}});

{ "\_id" : ObjectId("6181ee31a5580a8ff851c56d"), "name" : "suku1", "age" : 39, "exp" : 125 }

{ "\_id" : ObjectId("6181ee49a5580a8ff851c56e"), "name" : "suku1", "age" : 25, "exp" : 25 }

{ "\_id" : ObjectId("6181f282a5580a8ff851c56f"), "name" : "asuku1", "age" : 25, "exp" : 25 }

{ "\_id" : ObjectId("6181f288a5580a8ff851c570"), "name" : "asuku2", "age" : 25, "exp" : 25 }

c)$mod:- mod means modulo operator or remainder operator.It is very helpful to select documents based on modulo operation.

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

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

Note:- Both remainder and divisor should be provided otherwise we get error.

Example:-

> db.student.find()

{ "\_id" : ObjectId("6181ee25a5580a8ff851c56c"), "name" : "suku", "age" : 39, "exp" : 25 }

{ "\_id" : ObjectId("6181ee31a5580a8ff851c56d"), "name" : "suku1", "age" : 39, "exp" : 125 }

{ "\_id" : ObjectId("6181ee49a5580a8ff851c56e"), "name" : "suku1", "age" : 25, "exp" : 25 }

{ "\_id" : ObjectId("6181f282a5580a8ff851c56f"), "name" : "asuku1", "age" : 25, "exp" : 25 }

{ "\_id" : ObjectId("6181f288a5580a8ff851c570"), "name" : "asuku2", "age" : 25, "exp" : 25 }

> db.student.find({age:{$mod:[5,0]}});

{ "\_id" : ObjectId("6181ee49a5580a8ff851c56e"), "name" : "suku1", "age" : 25, "exp" : 25 }

{ "\_id" : ObjectId("6181f282a5580a8ff851c56f"), "name" : "asuku1", "age" : 25, "exp" : 25 }

{ "\_id" : ObjectId("6181f288a5580a8ff851c570"), "name" : "asuku2", "age" : 25, "exp" : 25 }

> db.student.find({age:{$mod:[5,1]}});

> db.student.find({age:{$mod:[5,2]}});

> db.student.find({age:{$mod:[5,3]}});

> db.student.find({age:{$mod:[5,4]}});

{ "\_id" : ObjectId("6181ee25a5580a8ff851c56c"), "name" : "suku", "age" : 39, "exp" : 25 }

{ "\_id" : ObjectId("6181ee31a5580a8ff851c56d"), "name" : "suku1", "age" : 39, "exp" : 125 }

6.2.6.5) **Array Query Operators**:- The array query operators are

a)$all b)$elemMatch c)$size.

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

Syntax:

-------

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

We can write equaivalent query by using $and operator also.

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

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

Example:-

------------

> db.student.find();

{ "\_id" : ObjectId("61820c1080214415365b489b"), "name" : "suku", "marks" : [ 10, 20, 30, 40 ] }

{ "\_id" : ObjectId("61820c2480214415365b489c"), "name" : "suku", "marks" : [ 10, 40, 70, 90 ] }

{ "\_id" : ObjectId("61820c2e80214415365b489d"), "name" : "suku1", "marks" : [ 50, 40, 170, 90 ] }

{ "\_id" : ObjectId("61820c3b80214415365b489e"), "name" : "suku2", "marks" : [ 150, 40, 170, 290 ] }

> db.student.find({marks:{$all:[10,20]}});

{ "\_id" : ObjectId("61820c1080214415365b489b"), "name" : "suku", "marks" : [ 10, 20, 30, 40 ] }

> db.student.find({marks:{$all:[40,90]}});

{ "\_id" : ObjectId("61820c2480214415365b489c"), "name" : "suku", "marks" : [ 10, 40, 70, 90 ] }

{ "\_id" : ObjectId("61820c2e80214415365b489d"), "name" : "suku1", "marks" : [ 50, 40, 170, 90 ] }

> db.student.find({marks:{$all:[90,40]}});

{ "\_id" : ObjectId("61820c2480214415365b489c"), "name" : "suku", "marks" : [ 10, 40, 70, 90 ] }

{ "\_id" : ObjectId("61820c2e80214415365b489d"), "name" : "suku1", "marks" : [ 50, 40, 170, 90 ] }

b**)$eleMatch**:- elemMatch means element Match.We can use $elemMatch operator to select documents where atleast one element of the array matches the specified query criteria.

Syntax:

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

Example:

> db.student.find();

{ "\_id" : ObjectId("61820c1080214415365b489b"), "name" : "suku", "marks" : [ 10, 20, 30, 40 ] }

{ "\_id" : ObjectId("61820c2480214415365b489c"), "name" : "suku", "marks" : [ 10, 40, 70, 90 ] }

{ "\_id" : ObjectId("61820c2e80214415365b489d"), "name" : "suku1", "marks" : [ 50, 40, 170, 90 ] }

{ "\_id" : ObjectId("61820c3b80214415365b489e"), "name" : "suku2", "marks" : [ 150, 40, 170, 290 ] }

> db.student.find({marks:{$gt:100,$lt:200}});

{ "\_id" : ObjectId("61820c2e80214415365b489d"), "name" : "suku1", "marks" : [ 50, 40, 170, 90 ] }

{ "\_id" : ObjectId("61820c3b80214415365b489e"), "name" : "suku2", "marks" : [ 150, 40, 170, 290 ] }

> db.student.find({marks:{$gt:100,$eq:200}});

c**)$size**:- It selects document in which the specified field exactly has specified size array.

Syntax:- {field:{$size: n}}

Example:-

--------------

db.student.find();

{ "\_id" : ObjectId("61820c1080214415365b489b"), "name" : "suku", "marks" : [ 10, 20, 30, 40 ] }

{ "\_id" : ObjectId("61820c2480214415365b489c"), "name" : "suku", "marks" : [ 10, 40, 70, 90 ] }

{ "\_id" : ObjectId("61820c2e80214415365b489d"), "name" : "suku1", "marks" : [ 50, 40, 170, 90 ] }

{ "\_id" : ObjectId("61820c3b80214415365b489e"), "name" : "suku2", "marks" : [ 150, 40, 170, 290 ] }

> db.student.find({marks:{$size:3}});

> db.student.find({marks:{$size:4}});

{ "\_id" : ObjectId("61820c1080214415365b489b"), "name" : "suku", "marks" : [ 10, 20, 30, 40 ] }

{ "\_id" : ObjectId("61820c2480214415365b489c"), "name" : "suku", "marks" : [ 10, 40, 70, 90 ] }

{ "\_id" : ObjectId("61820c2e80214415365b489d"), "name" : "suku1", "marks" : [ 50, 40, 170, 90 ] }

{ "\_id" : ObjectId("61820c3b80214415365b489e"), "name" : "suku2", "marks" : [ 150, 40, 170, 290 ] }

> db.student.find({marks:{$size:1}});

6.3)UpDate Record:- updation means

1. Overwrite existing value of a particular field with our new value

2. Add a new field for selected documents

3. Remove an existing field

4. Rename an existing field

etc

The update command is used to update or modify the documents in collections.

**Syntax:-1** db.collectionname.updateOne(selection\_criterial , update\_data,options);

It finds the first document that matches filter criteria and perform required updation. It will perform updation for a single document.

> db.mca.find();

{ "\_id" : ObjectId("61834e114c51886bf2376ad8"), "name" : "rock3", "age" : 35, "qual" : "mca" }

{ "\_id" : ObjectId("61834e214c51886bf2376ad9"), "name" : "rock1", "age" : 35, "qual" : "mba" }

{ "\_id" : ObjectId("61834e2e4c51886bf2376ada"), "name" : "rock2", "age" : 37, "qual" : "mcom" }

{ "\_id" : ObjectId("61834e454c51886bf2376adb"), "name" : "rock1", "age" : 39, "qual" : "bca" }

{ "\_id" : ObjectId("61834e5c4c51886bf2376adc"), "name" : "suku", "age" : 40, "qual" : "M.tech" }

> db.mca.updateOne({name:'rock1'},{'age':47});

> db.mca.updateOne({name:'rock1'},{$set:{'age':47}});

{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }

> db.mca.find();

{ "\_id" : ObjectId("61834e114c51886bf2376ad8"), "name" : "rock3", "age" : 35, "qual" : "mca" }

{ "\_id" : ObjectId("61834e214c51886bf2376ad9"), "name" : "rock1", "age" : 47, "qual" : "mba" }

{ "\_id" : ObjectId("61834e2e4c51886bf2376ada"), "name" : "rock2", "age" : 37, "qual" : "mcom" }

{ "\_id" : ObjectId("61834e454c51886bf2376adb"), "name" : "rock1", "age" : 39, "qual" : "bca" }

{ "\_id" : ObjectId("61834e5c4c51886bf2376adc"), "name" : "suku", "age" : 40, "qual" : "M.tech" }

**Syntax:-**2 db.collectionname.updateMany(selection\_criterial , update\_data,options);

To update all documents that match the specified filter criteria.

> db.mca.updateMany({name:'rock1'},{$set:{age:57}});

{ "acknowledged" : true, "matchedCount" : 2, "modifiedCount" : 2 }

> db.mca.find();

{ "\_id" : ObjectId("61834e114c51886bf2376ad8"), "name" : "rock3", "age" : 35, "qual" : "mca" }

{ "\_id" : ObjectId("61834e214c51886bf2376ad9"), "name" : "rock1", "age" : 57, "qual" : "mba" }

{ "\_id" : ObjectId("61834e2e4c51886bf2376ada"), "name" : "rock2", "age" : 37, "qual" : "mcom" }

{ "\_id" : ObjectId("61834e454c51886bf2376adb"), "name" : "rock1", "age" : 57, "qual" : "bca" }

{ "\_id" : ObjectId("61834e5c4c51886bf2376adc"), "name" : "suku", "age" : 40, "qual" : "M.tech" }

>

**Syntax:3** db.collectionname.updateMany(selection\_criterial , update\_data,options);

We can use this method to update either a single document or multiple documents.Bydefault this method updates a single document only.If we include multi:true to update all documents that match query criteria.

db.collection.update(filter,update)--->To update a single document

db.collection.update(filter,update,{multi:true})--->To update all matched documents .

> db.mca.find();

{ "\_id" : ObjectId("61834e114c51886bf2376ad8"), "name" : "rock3", "age" : 35, "qual" : "mca" }

{ "\_id" : ObjectId("61834e214c51886bf2376ad9"), "name" : "rock1", "age" : 57, "qual" : "mba" }

{ "\_id" : ObjectId("61834e2e4c51886bf2376ada"), "name" : "rock2", "age" : 37, "qual" : "mcom" }

{ "\_id" : ObjectId("61834e454c51886bf2376adb"), "name" : "rock1", "age" : 57, "qual" : "bca" }

{ "\_id" : ObjectId("61834e5c4c51886bf2376adc"), "name" : "suku", "age" : 40, "qual" : "M.tech" }

> db.mca.update({name:'rock1'},{$set:{age:59}});

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

> db.mca.find();

{ "\_id" : ObjectId("61834e114c51886bf2376ad8"), "name" : "rock3", "age" : 35, "qual" : "mca" }

{ "\_id" : ObjectId("61834e214c51886bf2376ad9"), "name" : "rock1", "age" : 59, "qual" : "mba" }

{ "\_id" : ObjectId("61834e2e4c51886bf2376ada"), "name" : "rock2", "age" : 37, "qual" : "mcom" }

{ "\_id" : ObjectId("61834e454c51886bf2376adb"), "name" : "rock1", "age" : 57, "qual" : "bca" }

{ "\_id" : ObjectId("61834e5c4c51886bf2376adc"), "name" : "suku", "age" : 40, "qual" : "M.tech" }

> db.mca.update({name:'rock1'},{$set:{age:60}},{multi:true});

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

> db.mca.find()

{ "\_id" : ObjectId("61834e114c51886bf2376ad8"), "name" : "rock3", "age" : 35, "qual" : "mca" }

{ "\_id" : ObjectId("61834e214c51886bf2376ad9"), "name" : "rock1", "age" : 60, "qual" : "mba" }

{ "\_id" : ObjectId("61834e2e4c51886bf2376ada"), "name" : "rock2", "age" : 37, "qual" : "mcom" }

{ "\_id" : ObjectId("61834e454c51886bf2376adb"), "name" : "rock1", "age" : 60, "qual" : "bca" }

{ "\_id" : ObjectId("61834e5c4c51886bf2376adc"), "name" : "suku", "age" : 40, "qual" : "M.tech" }

6.3.1)Update Operators:- We can use update operators to perform required updations.

1. $set

2. $unset

3. $rename

4. $inc

5. $min

6. $max

7. $mul

Etc.

6.3.1.a) $set:- We can use $set operator to set the value to the field in matched document.

Note:1 If the specified field does not exist, $set will add a new field with provided value.

2.If the query-criteria is not in update then new field will be added to all documents.

Syntax:1 {$set:{field1:value}}  
 Syntax:-2 {$set:{field1:value,field2;value,…}

Example:-

> db.mca.update({},{$set:{'town':'nlr'}});

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

> db.mca.find();

{ "\_id" : ObjectId("61834e114c51886bf2376ad8"), "name" : "rock3", "age" : 35, "qual" : "mca", "town" : "nlr" }

{ "\_id" : ObjectId("61834e214c51886bf2376ad9"), "name" : "rock1", "age" : 60, "qual" : "mba" }

{ "\_id" : ObjectId("61834e2e4c51886bf2376ada"), "name" : "rock2", "age" : 37, "qual" : "mcom" }

{ "\_id" : ObjectId("61834e454c51886bf2376adb"), "name" : "rock1", "age" : 60, "qual" : "bca" }

{ "\_id" : ObjectId("61834e5c4c51886bf2376adc"), "name" : "suku", "age" : 40, "qual" : "M.tech" }

> db.mca.updateMany({},{$set:{'town':'nlr'}});

{ "acknowledged" : true, "matchedCount" : 5, "modifiedCount" : 4 }

> db.mca.find();

{ "\_id" : ObjectId("61834e114c51886bf2376ad8"), "name" : "rock3", "age" : 35, "qual" : "mca", "town" : "nlr" }

{ "\_id" : ObjectId("61834e214c51886bf2376ad9"), "name" : "rock1", "age" : 60, "qual" : "mba", "town" : "nlr" }

{ "\_id" : ObjectId("61834e2e4c51886bf2376ada"), "name" : "rock2", "age" : 37, "qual" : "mcom", "town" : "nlr" }

{ "\_id" : ObjectId("61834e454c51886bf2376adb"), "name" : "rock1", "age" : 60, "qual" : "bca", "town" : "nlr" }

{ "\_id" : ObjectId("61834e5c4c51886bf2376adc"), "name" : "suku", "age" : 40, "qual" : "M.tech", "town" : "nlr" }

>

6.3.1.b) unSet:- To delete specified field.

Syntax:- db.collectionname.update({filter/query},{$unset:{field1: “ “,field2:” “,…}});

The specified value in the $unset expression (ie "") does not impact operation. If the specified field is not avaialble, then $unset operator won't do anything. Example:

> db.mca.find();

{ "\_id" : ObjectId("61834e114c51886bf2376ad8"), "name" : "rock3", "age" : 35, "qual" : "mca", "town" : "nlr" }

{ "\_id" : ObjectId("61834e214c51886bf2376ad9"), "name" : "rock1", "age" : 60, "qual" : "mba", "town" : "nlr" }

{ "\_id" : ObjectId("61834e2e4c51886bf2376ada"), "name" : "rock2", "age" : 37, "qual" : "mcom", "town" : "nlr" }

{ "\_id" : ObjectId("61834e454c51886bf2376adb"), "name" : "rock1", "age" : 60, "qual" : "bca", "town" : "nlr" }

{ "\_id" : ObjectId("61834e5c4c51886bf2376adc"), "name" : "suku", "age" : 40, "qual" : "M.tech", "town" : "nlr" }

> db.mca.updateMany({},{$unset:{'town':" "}});

{ "acknowledged" : true, "matchedCount" : 5, "modifiedCount" : 5 }

> db.mca.find();

{ "\_id" : ObjectId("61834e114c51886bf2376ad8"), "name" : "rock3", "age" : 35, "qual" : "mca" }

{ "\_id" : ObjectId("61834e214c51886bf2376ad9"), "name" : "rock1", "age" : 60, "qual" : "mba" }

{ "\_id" : ObjectId("61834e2e4c51886bf2376ada"), "name" : "rock2", "age" : 37, "qual" : "mcom" }

{ "\_id" : ObjectId("61834e454c51886bf2376adb"), "name" : "rock1", "age" : 60, "qual" : "bca" }

{ "\_id" : ObjectId("61834e5c4c51886bf2376adc"), "name" : "suku", "age" : 40, "qual" : "M.tech" }

>

6.3.1.c)rename:- We can use $rename operator to rename fields, ie to change name of the field.

Syntax:

{$rename: {field1:<newName1>, field2:<newName2>, ...} }

Note:

-----

1. The $rename operator internally performs $unset of both old name and new name and then performs $set with new name. Hence it won't preserve order of fields.

2. If the document already has a field with newName then $rename operator removes that field and enames specified field with newName.

3. If the field to rename does not exist in the document then $rename won't do anything.

Example:-

-------------

> db.mca.find();

{ "\_id" : ObjectId("61834e114c51886bf2376ad8"), "name" : "rock3", "age" : 35, "qual" : "mca" }

{ "\_id" : ObjectId("61834e214c51886bf2376ad9"), "name" : "rock1", "age" : 60, "qual" : "mba" }

{ "\_id" : ObjectId("61834e2e4c51886bf2376ada"), "name" : "rock2", "age" : 37, "qual" : "mcom" }

{ "\_id" : ObjectId("61834e454c51886bf2376adb"), "name" : "rock1", "age" : 60, "qual" : "bca" }

{ "\_id" : ObjectId("61834e5c4c51886bf2376adc"), "name" : "suku", "age" : 40, "qual" : "M.tech" }

> db.mca.updateMany({name:'rock1'},{$rename:{qual:'qualification'}});

{ "acknowledged" : true, "matchedCount" : 2, "modifiedCount" : 2 }

> db.mca.find();

{ "\_id" : ObjectId("61834e114c51886bf2376ad8"), "name" : "rock3", "age" : 35, "qual" : "mca" }

{ "\_id" : ObjectId("61834e214c51886bf2376ad9"), "name" : "rock1", "age" : 60, "qualification" : "mba" }

{ "\_id" : ObjectId("61834e2e4c51886bf2376ada"), "name" : "rock2", "age" : 37, "qual" : "mcom" }

{ "\_id" : ObjectId("61834e454c51886bf2376adb"), "name" : "rock1", "age" : 60, "qualification" : "bca" }

{ "\_id" : ObjectId("61834e5c4c51886bf2376adc"), "name" : "suku", "age" : 40, "qual" : "M.tech" }

> db.mca.updateMany({name:'rock1'},{$rename:{qualification:'age'}});

{ "acknowledged" : true, "matchedCount" : 2, "modifiedCount" : 2 }

> db.mca.find();

{ "\_id" : ObjectId("61834e114c51886bf2376ad8"), "name" : "rock3", "age" : 35, "qual" : "mca" }

{ "\_id" : ObjectId("61834e214c51886bf2376ad9"), "name" : "rock1", "age" : "mba" }

{ "\_id" : ObjectId("61834e2e4c51886bf2376ada"), "name" : "rock2", "age" : 37, "qual" : "mcom" }

{ "\_id" : ObjectId("61834e454c51886bf2376adb"), "name" : "rock1", "age" : "bca" }

{ "\_id" : ObjectId("61834e5c4c51886bf2376adc"), "name" : "suku", "age" : 40, "qual" : "M.tech" }

6.3.1.d) $inc:- inc means increment.

We can use $inc to increment or decrement value of the field with specified amount.

Syntax:

{$inc: {field1:amount1,field2:amount,..}}

$inc can take both positive and negative values.

positive value for increment operation.

negative value for decrement operation.

Note:1 If the specified field does not exist, $inc creates that field and sets that field to the specified value.

2. We cannot perform multiple updates on the same field at a time, otherwise we will get error.

> db.mca.update({age:35},{$inc:{age:20}});

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

> db.mca.find();

{ "\_id" : ObjectId("61834e114c51886bf2376ad8"), "name" : "rock3", "age" : 55, "qual" : "mca" }

{ "\_id" : ObjectId("61834e214c51886bf2376ad9"), "name" : "rock1", "age" : "mba" }

{ "\_id" : ObjectId("61834e2e4c51886bf2376ada"), "name" : "rock2", "age" : 37, "qual" : "mcom" }

{ "\_id" : ObjectId("61834e454c51886bf2376adb"), "name" : "rock1", "age" : "bca" }

{ "\_id" : ObjectId("61834e5c4c51886bf2376adc"), "name" : "suku", "age" : 40, "qual" : "M.tech" }

> db.mca.update({age:55},{$inc:{age:-20}});

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

> db.mca.find()

{ "\_id" : ObjectId("61834e114c51886bf2376ad8"), "name" : "rock3", "age" : 35, "qual" : "mca" }

{ "\_id" : ObjectId("61834e214c51886bf2376ad9"), "name" : "rock1", "age" : "mba" }

{ "\_id" : ObjectId("61834e2e4c51886bf2376ada"), "name" : "rock2", "age" : 37, "qual" : "mcom" }

{ "\_id" : ObjectId("61834e454c51886bf2376adb"), "name" : "rock1", "age" : "bca" }

{ "\_id" : ObjectId("61834e5c4c51886bf2376adc"), "name" : "suku", "age" : 40, "qual" : "M.tech" }

>

6.3.1.e)$min:- It only updates field value if the specified value is less than current field value .

Note1: If the specified field does not exist, then $min operator createsthat field and assign with provided value:

Syntax:- 1 {$min: {field1:value1, field2:value2,...}}

Syntax:-2 {$min: {field1:value1}}

Example:-

> db.mca.find();

{ "\_id" : ObjectId("61834e114c51886bf2376ad8"), "name" : "rock3", "age" : 35, "qual" : "mca" }

{ "\_id" : ObjectId("61834e214c51886bf2376ad9"), "name" : "rock1", "age" : 45 }

{ "\_id" : ObjectId("61834e2e4c51886bf2376ada"), "name" : "rock2", "age" : 37, "qual" : "mcom" }

{ "\_id" : ObjectId("61834e454c51886bf2376adb"), "name" : "rock1", "age" : 45 }

{ "\_id" : ObjectId("61834e5c4c51886bf2376adc"), "name" : "suku", "age" : 40, "qual" : "M.tech" }

> db.mca.updateMany({name:'rock1'},{$min:{age:42}});

{ "acknowledged" : true, "matchedCount" : 2, "modifiedCount" : 2 }

> db.mca.find();

{ "\_id" : ObjectId("61834e114c51886bf2376ad8"), "name" : "rock3", "age" : 35, "qual" : "mca" }

{ "\_id" : ObjectId("61834e214c51886bf2376ad9"), "name" : "rock1", "age" : 42 }

{ "\_id" : ObjectId("61834e2e4c51886bf2376ada"), "name" : "rock2", "age" : 37, "qual" : "mcom" }

{ "\_id" : ObjectId("61834e454c51886bf2376adb"), "name" : "rock1", "age" : 42 }

{ "\_id" : ObjectId("61834e5c4c51886bf2376adc"), "name" : "suku", "age" : 40, "qual" : "M.tech" }

6.3.1.f)$max:- $max operator updates the value of the field to the specified value iff

specified value is greater than current value.

Syntax:- 1 {$max: {field1:value1, field2:value2,...}}

Syntax:-2 {$max: {field1:value1}}

Note1: If the specified field does not exist, then $min operator creates that field and assign with provided value.

6.3.1.g) $mul:- mul means multiplication.We can use $mul operator to multiply the value of a field by a number.

{$mul: {field: number}}

The field to update must contain numeric value.

6.3.2) Upsert Command:- Whenever we are trying to perform update operation, the matched

document may or may not available.If it is available then it will be updated and if it is not available then update won't be happend.If the document not available then we can insert that document in the database automatically. For this we have to use upsert property.146

upsert = update + insert

at the time two works update and insert. First update and if it is not possible then insert.

upsert will take boolean value.

If it is set to true, it will creates a new document if it is not available.

If it is set to false, then it will perform just update operation and won't

create any new document.

The default value of upsert is: false.

Example:-

------------

> db.mca.update({name:'suku1'},{$set:{name:'suku1','age':40,'qual':'MCA'}},{upsert:true});

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

> db.mca.find();

{ "\_id" : ObjectId("61834e114c51886bf2376ad8"), "name" : "rock3", "age" : 35, "qual" : "mca" }

{ "\_id" : ObjectId("61834e214c51886bf2376ad9"), "name" : "rock1", "age" : 42 }

{ "\_id" : ObjectId("61834e2e4c51886bf2376ada"), "name" : "rock2", "age" : 37, "qual" : "mcom" }

{ "\_id" : ObjectId("61834e454c51886bf2376adb"), "name" : "rock1", "age" : 42 }

{ "\_id" : ObjectId("61834e5c4c51886bf2376adc"), "name" : "suku", "age" : 40, "qual" : "M.tech" }

{ "\_id" : ObjectId("61837a7e3e553ac66004caa7"), "name" : "suku1", "age" : 40, "qual" : "MCA" }

> db.mca.update({name:'suku3'},{$set:{name:'suku1','age':40,'qual':'MCA'}},{upsert:true});

WriteResult({

"nMatched" : 0,

"nUpserted" : 1,

"nModified" : 0,

"\_id" : ObjectId("61837ac33e553ac66004cab2")

})

> db.mca.find();

{ "\_id" : ObjectId("61834e114c51886bf2376ad8"), "name" : "rock3", "age" : 35, "qual" : "mca" }

{ "\_id" : ObjectId("61834e214c51886bf2376ad9"), "name" : "rock1", "age" : 42 }

{ "\_id" : ObjectId("61834e2e4c51886bf2376ada"), "name" : "rock2", "age" : 37, "qual" : "mcom" }

{ "\_id" : ObjectId("61834e454c51886bf2376adb"), "name" : "rock1", "age" : 42 }

{ "\_id" : ObjectId("61834e5c4c51886bf2376adc"), "name" : "suku", "age" : 40, "qual" : "M.tech" }

{ "\_id" : ObjectId("61837a7e3e553ac66004caa7"), "name" : "suku1", "age" : 40, "qual" : "MCA" }

{ "\_id" : ObjectId("61837ac33e553ac66004cab2"), "name" : "suku1", "age" : 40, "qual" : "MCA" }

>

6.3.3) Array Update operators:- To update the value of Array in document, we use one of following operatators.

a) $ b) $[] c) $[<identifier>]

a)$:- $ acts as a placeholder to update first matched element based on query condition.

Syntax:

db.collection.update(query-condition,{update\_operator:{"<array>.$" : value}})

The array field must appear as the part of query condition.

Examples:-

> db.mca.find();

{ "\_id" : ObjectId("6183a5df66de04c1e59c9c94"), "name" : "suku", "age" : [ 1, 2, 3, 4 ] }

{ "\_id" : ObjectId("6183a5ea66de04c1e59c9c95"), "name" : "suku1", "age" : [ 11, 12, 13, 14 ] }

{ "\_id" : ObjectId("6183a5f666de04c1e59c9c96"), "name" : "suku2", "age" : [ 11, 22, 23, 34 ] }

> db.mca.update({age:2},{$set:{'age.$':72}});

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

> db.mca.find();

{ "\_id" : ObjectId("6183a5df66de04c1e59c9c94"), "name" : "suku", "age" : [ 1, 72, 3, 4 ] }

{ "\_id" : ObjectId("6183a5ea66de04c1e59c9c95"), "name" : "suku1", "age" : [ 11, 12, 13, 14 ] }

{ "\_id" : ObjectId("6183a5f666de04c1e59c9c96"), "name" : "suku2", "age" : [ 11, 22, 23, 34 ] }

> db.mca.update({age:{$elemMatch:{$lt:15}}},{$set:{'age.$':77}});

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

> db.mca.find();

{ "\_id" : ObjectId("6183a5df66de04c1e59c9c94"), "name" : "suku", "age" : [ 77, 72, 3, 4 ] }

{ "\_id" : ObjectId("6183a5ea66de04c1e59c9c95"), "name" : "suku1", "age" : [ 11, 12, 13, 14 ] }

{ "\_id" : ObjectId("6183a5f666de04c1e59c9c96"), "name" : "suku2", "age" : [ 11, 22, 23, 34 ] }

b)$[]:- $[] acts as placeholder to update all elements in the array for the matched documents based on query condition.

Syntax:

db.collection.update(query,{update\_operator:{"<array>.$[]" : value}})

Ex:-

> db.mca.find();

{ "\_id" : ObjectId("6183a5df66de04c1e59c9c94"), "name" : "suku", "age" : [ 77, 72, 3, 4 ] }

{ "\_id" : ObjectId("6183a5ea66de04c1e59c9c95"), "name" : "suku1", "age" : [ 11, 12, 13, 14 ] }

{ "\_id" : ObjectId("6183a5f666de04c1e59c9c96"), "name" : "suku2", "age" : [ 11, 22, 23, 34 ] }

> db.mca.updateMany({name:'suku1'},{$set:{'age.$[]':77}});

{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }

> db.mca.find();

{ "\_id" : ObjectId("6183a5df66de04c1e59c9c94"), "name" : "suku", "age" : [ 77, 72, 3, 4 ] }

{ "\_id" : ObjectId("6183a5ea66de04c1e59c9c95"), "name" : "suku1", "age" : [ 77, 77, 77, 77 ] }

{ "\_id" : ObjectId("6183a5f666de04c1e59c9c96"), "name" : "suku2", "age" : [ 11, 22, 23, 34 ] }

> db.mca.updateMany({},{$set:{'age.$[]':77}});

{ "acknowledged" : true, "matchedCount" : 3, "modifiedCount" : 2 }

> db.mca.find();

{ "\_id" : ObjectId("6183a5df66de04c1e59c9c94"), "name" : "suku", "age" : [ 77, 77, 77, 77 ] }

{ "\_id" : ObjectId("6183a5ea66de04c1e59c9c95"), "name" : "suku1", "age" : [ 77, 77, 77, 77 ] }

{ "\_id" : ObjectId("6183a5f666de04c1e59c9c96"), "name" : "suku2", "age" : [ 77, 77, 77, 77 ] }

c)$[<identifier>]:-Instead of updating only first matched element or all elements of the array, we can update only required array elements.

For this we have to use $[identifier]

$[identifier] --->Acts as placeholder to update all elements that match the arrayFilters condition for the documents that match query condition.

In this case updation is based on arrayFilters condition but not based on query condition.

Syntax:1 db.collectionname.updateMany({},{update-operator:{fieldname.$[<identifier>]: value}},{arrayFilter: [filter1]});

Syntax:2 db.coll-name.updateMany({},{update-opertor:{fieldname.$[<identifier>]:value,fieldname.$[<identifier>]:value,…}},{arrayFilter:[filter1,filter2….]});

Example:

> db.mca.updateMany({},{$set:{'age.$[x]':30}},{arrayFilters:[{$and:[{'x':{$gt:11}},{'x':{$lt:14}}]}]});

{ "acknowledged" : true, "matchedCount" : 3, "modifiedCount" : 1 }

> db.mca.find();

{ "\_id" : ObjectId("6183ac2a66de04c1e59c9c97"), "name" : "suku", "age" : [ 30, 30, 3, 4 ] }

{ "\_id" : ObjectId("6183ac3966de04c1e59c9c98"), "name" : "suku1", "age" : [ 11, 30, 30, 14 ] }

{ "\_id" : ObjectId("6183ac4766de04c1e59c9c99"), "name" : "suku2", "age" : [ 21, 22, 23, 24 ] }

>

6.3.4) Adding the elements to Array:-

**A)$push**:-we can Adding elements to the array by using $push operator.We can use $push operator to add elements to the array. By default element will be added at the end, but based on our requirement we can add in our required position also.

Syntax: db.collection.update({},{$push: {<array1>: value1,...}})

Example:

-----------

> db.mca.find();

{ "\_id" : ObjectId("6183ac2a66de04c1e59c9c97"), "name" : "suku", "age" : [ 30, 30, 3, 4 ] }

{ "\_id" : ObjectId("6183ac3966de04c1e59c9c98"), "name" : "suku1", "age" : [ 11, 30, 30, 14 ] }

{ "\_id" : ObjectId("6183ac4766de04c1e59c9c99"), "name" : "suku2", "age" : [ 21, 22, 23, 24 ] }

> db.mca.updateMany({},{$push:{age:38}});

{ "acknowledged" : true, "matchedCount" : 3, "modifiedCount" : 3 }

> db.mca.find();

{ "\_id" : ObjectId("6183ac2a66de04c1e59c9c97"), "name" : "suku", "age" : [ 30, 30, 3, 4, 38 ] }

{ "\_id" : ObjectId("6183ac3966de04c1e59c9c98"), "name" : "suku1", "age" : [ 11, 30, 30, 14, 38 ] }

{ "\_id" : ObjectId("6183ac4766de04c1e59c9c99"), "name" : "suku2", "age" : [ 21, 22, 23, 24, 38 ] }

Example2:--

> db.mca.find();

{ "\_id" : ObjectId("6183ac2a66de04c1e59c9c97"), "name" : "suku", "age" : [ 30, 30, 3, 4, 38 ] }

{ "\_id" : ObjectId("6183ac3966de04c1e59c9c98"), "name" : "suku1", "age" : [ 11, 30, 30, 14, 38 ] }

{ "\_id" : ObjectId("6183ac4766de04c1e59c9c99"), "name" : "suku2", "age" : [ 21, 22, 23, 24, 38 ] }

> db.mca.updateMany({},{$push:{age:39,age:40,age:41}});

{ "acknowledged" : true, "matchedCount" : 3, "modifiedCount" : 3 }

> db.mca.find();

{ "\_id" : ObjectId("6183ac2a66de04c1e59c9c97"), "name" : "suku", "age" : [ 30, 30, 3, 4, 38, 41 ] }

{ "\_id" : ObjectId("6183ac3966de04c1e59c9c98"), "name" : "suku1", "age" : [ 11, 30, 30, 14, 38, 41 ] }

{ "\_id" : ObjectId("6183ac4766de04c1e59c9c99"), "name" : "suku2", "age" : [ 21, 22, 23, 24, 38, 41 ] }

>

Note:- Reason: In Javascript object, duplicate keys are not allowed. If we aretrying to add entry with duplicate key, old value will be replaced with new value.

B**)$each modifier**:- We can use $each modifier to add multiple values to the array.

Syntax:{ $push: { <array>: {$each: [value1,value2,...]} }

Example:-

> db.mca.find();

{ "\_id" : ObjectId("6183ac2a66de04c1e59c9c97"), "name" : "suku", "age" : [ 30, 30, 3, 4, 38, 41 ] }

{ "\_id" : ObjectId("6183ac3966de04c1e59c9c98"), "name" : "suku1", "age" : [ 11, 30, 30, 14, 38, 41 ] }

{ "\_id" : ObjectId("6183ac4766de04c1e59c9c99"), "name" : "suku2", "age" : [ 21, 22, 23, 24, 38, 41 ] }

> db.mca.updateMany({},{$push:{age:{$each:[39,40,41]}}});

{ "acknowledged" : true, "matchedCount" : 3, "modifiedCount" : 3 }

> db.mca.find();

{ "\_id" : ObjectId("6183ac2a66de04c1e59c9c97"), "name" : "suku", "age" : [ 30, 30, 3, 4, 38, 41, 39, 40, 41 ] }

{ "\_id" : ObjectId("6183ac3966de04c1e59c9c98"), "name" : "suku1", "age" : [ 11, 30, 30, 14, 38, 41, 39, 40, 41 ] }

{ "\_id" : ObjectId("6183ac4766de04c1e59c9c99"), "name" : "suku2", "age" : [ 21, 22, 23, 24, 38, 41, 39, 40, 41 ] }

**c)$position**: Bydefault elements will be added at the end of the array. But we can add elements in the required position. For this we have to use $position modifier.

To use $position modifier, compulsory we should use $each modifier.

i.e $position without $each is always invalid.

Syntax:

{

$push: {

<array>: {

$each: [value1,value2,value3],

$position: <num>

}

}

}

<num> indicates the position where we have to add element. Array follows zero based index. ie index of first element is 0. If <num> is negative, index starts from ending.

If <num> is greather than size of array then elements are added at end of the array.

Examples:-

------------

> db.mca.updateMany({},{$push:{age:{$each:[777],$position:0}}});

{ "acknowledged" : true, "matchedCount" : 3, "modifiedCount" : 3 }

> db.mca.find();

{ "\_id" : ObjectId("6183ac2a66de04c1e59c9c97"), "name" : "suku", "age" : [ 777, 30, 30, 3, 4, 38, 41, 39, 40, 41 ] }

{ "\_id" : ObjectId("6183ac3966de04c1e59c9c98"), "name" : "suku1", "age" : [ 777, 11, 30, 30, 14, 38, 41, 39, 40, 41 ] }

{ "\_id" : ObjectId("6183ac4766de04c1e59c9c99"), "name" : "suku2", "age" : [ 777, 21, 22, 23, 24, 38, 41, 39, 40, 41 ] }

>

d**)$sort modifier**:- We can use $sort modifier to sort elements of the array while

performing push operation. To use $sort modifier, we should use $each modifier. i.e without

$each, we cannot use $sort modifier.

We can pass empty array [], to $each modifier to see effect of only

$sort.

Syntax:

-------

db.collection.update({},

{

$push: {

<array>: { $each: [value1,value2,..],

$sort: 1|-1

}

}

})

1 means ascending order

-1 means descending order

Examples:-

> db.mca.find()

{ "\_id" : ObjectId("618496e6a3e09f155ac2f769"), "name" : "suku", "age" : [ 1, 2, 3, 4 ] }

{ "\_id" : ObjectId("61849719a3e09f155ac2f76a"), "name" : "suku1", "age" : [ 11, 12, 23, 14 ] }

{ "\_id" : ObjectId("61849728a3e09f155ac2f76b"), "name" : "suku2", "age" : [ 21, 22, 24, 34 ] }

}

})

> db.mca.update({},{$push:{age:{$each:[77,78],$sort:-1}}});

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

> db.mca.find()

{ "\_id" : ObjectId("618496e6a3e09f155ac2f769"), "name" : "suku", "age" : [ 78, 77, 4, 3, 2, 1 ] }

{ "\_id" : ObjectId("61849719a3e09f155ac2f76a"), "name" : "suku1", "age" : [ 11, 12, 23, 14 ] }

{ "\_id" : ObjectId("61849728a3e09f155ac2f76b"), "name" : "suku2", "age" : [ 21, 22, 24, 34 ] }

> db.mca.updateMany({},{$push:{age:{$each:[],$sort:-1}}});

{ "acknowledged" : true, "matchedCount" : 3, "modifiedCount" : 3 }

> db.mca.find()

{ "\_id" : ObjectId("618496e6a3e09f155ac2f769"), "name" : "suku", "age" : [ 78, 77, 4, 3, 2, 1 ] }

{ "\_id" : ObjectId("61849719a3e09f155ac2f76a"), "name" : "suku1", "age" : [ 23, 14, 12, 11 ] }

{ "\_id" : ObjectId("61849728a3e09f155ac2f76b"), "name" : "suku2", "age" : [ 34, 24, 22, 21 ] }

e)$slice Modifier:- The $slice modifier limits the number of array elements during $push

operation. To use $slice modifier, we should use $each modifier. i.e without $each, we cannot use $slice modifier.We can pass empty array [], to $each modifier to see effect of only $slice modifier.

Syntax:

------

db.collection.update(

{},

{

$push: {

<array>: { $each: [value1,value2,..],

$slice: <num>

}

}

})

The <num> can be:

1. zero --->To update array to an empty array.

2. positive --->To update array field to contain only first <num> elements.

3. Negative --->To update array field to contain only last <num> elements.

But MongoDB Server will process push operation in the following order:

1. Update array to add elements in the correct position.

2. Apply sort, if specified.

3. slice the array, if specified.

4. Store the array.

Example:-

> db.mca.find()

{ "\_id" : ObjectId("618496e6a3e09f155ac2f769"), "name" : "suku", "age" : [ 78, 77, 4, 3, 2, 1 ] }

{ "\_id" : ObjectId("61849719a3e09f155ac2f76a"), "name" : "suku1", "age" : [ 23, 14, 12, 11 ] }

{ "\_id" : ObjectId("61849728a3e09f155ac2f76b"), "name" : "suku2", "age" : [ 34, 24, 22, 21 ] }

> db.mca.update({name:'suku'},{$push:{age:{$each:[99,89],$slice:1}}});

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

> db.mca.find()

{ "\_id" : ObjectId("618496e6a3e09f155ac2f769"), "name" : "suku", "age" : [ 78 ] }

{ "\_id" : ObjectId("61849719a3e09f155ac2f76a"), "name" : "suku1", "age" : [ 23, 14, 12, 11 ] }

{ "\_id" : ObjectId("61849728a3e09f155ac2f76b"), "name" : "suku2", "age" : [ 34, 24, 22, 21 ] }

> db.mca.update({name:'suku'},{$push:{age:{$each:[99,89],$slice:3}}});

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

> db.mca.find()

{ "\_id" : ObjectId("618496e6a3e09f155ac2f769"), "name" : "suku", "age" : [ 78, 99, 89 ] }

{ "\_id" : ObjectId("61849719a3e09f155ac2f76a"), "name" : "suku1", "age" : [ 23, 14, 12, 11 ] }

{ "\_id" : ObjectId("61849728a3e09f155ac2f76b"), "name" : "suku2", "age" : [ 34, 24, 22, 21 ] }

> db.mca.update({name:'suku'},{$push:{age:{$each:[199,189,200],$slice:6,$sort:1}}});

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

> db.mca.find()

{ "\_id" : ObjectId("618496e6a3e09f155ac2f769"), "name" : "suku", "age" : [ 78, 89, 99, 189, 199, 200 ] }

{ "\_id" : ObjectId("61849719a3e09f155ac2f76a"), "name" : "suku1", "age" : [ 23, 14, 12, 11 ] }

{ "\_id" : ObjectId("61849728a3e09f155ac2f76b"), "name" : "suku2", "age" : [ 34, 24, 22, 21 ] }

1. If the spcified array is not already available then $push adds that

array field with values as its elements.

> db.students.update({\_id:7},{$push: {marks1: {$each: [10,20,30]}}})

{ "\_id" : 7, "marks" : [ 9, 8, 7 ], "marks1" : [ 10, 20, 30 ] }

2. If the field is not array, then $push operation will fail.

> db.students.update({\_id:7},{$set: {name:"Durga"}})

{ "\_id" : 7, "marks" : [ 9, 8, 7 ], "marks1" : [ 10, 20, 30 ], "name" :

"Durga" }

> db.students.update({\_id:7},{$push: {name:{$each: [10,20,30]}}})

"errmsg" : "The field 'name' must be an array but is of type string in

document {\_id: 7.0}"

6.3.5)$addToSet Operators:- It is exactly same as $push operator except that it won't allow

duplicates.It adds elements to the array iff array does not contain already those elements.

There is no effect on already existing duplicates.

Notes:1: In the case of $push operator, order terminology is applicable. Hence we can use $position, $sort, $slice modifiers.But in the case of $addToSet operator, order terminology is not applicable. Hence we cannot use $position, $sort, $slice modifiers.But $each modifier applicable for both $push and $addToSet operators.

Syntax:

------

db.collection.update(

{},

{

$addToSet: {

<array>: { $each: [value1,value2,..],

}

})

Example:-

> db.mca.find()

{ "\_id" : ObjectId("61849728a3e09f155ac2f76b"), "name" : "suku2", "age" : [ 34, 24, 22, 21 ] }

> db.mca.update({name:'suku2'},{$addToSet:{age:{$each:[45,34,78,89]}}});

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

> db.mca.find()

{ "\_id" : ObjectId("61849728a3e09f155ac2f76b"), "name" : "suku2", "age" : [ 34, 24, 22, 21, 45, 78, 89 ] }

> db.students.update({\_id:5},{$addToSet: {marks: {$each: [7,8,9],

$position: 2}}})

"errmsg" : "Found unexpected fields after $each in $addToSet: {

$each: [ 7.0, 8.0, 9.0 ], $position: 2.0 }"

6.3.6)Removing the elements from array using $pop:- We can use $pop operator to remove either first or last element from the array.

Syntax:

------

{

$pop: {<array>:-1|1}

}

-1 --->To remove the first element

1 --->To remove the last element

Example:-

-----------

> db.mca.find();

{ "\_id" : ObjectId("6184ab1ca3e09f155ac2f76c"), "name" : "suku", "age" : [ 1, 2, 3, 4 ] }

> db.mca.update({'name':'suku'},{$pop:{age:1}});

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

> db.mca.find();

{ "\_id" : ObjectId("6184ab1ca3e09f155ac2f76c"), "name" : "suku", "age" : [ 1, 2, 3 ] }

> db.mca.update({'name':'suku'},{$pop:{age:-1}});

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

> db.mca.find();

{ "\_id" : ObjectId("6184ab1ca3e09f155ac2f76c"), "name" : "suku", "age" : [ 2, 3 ] }

6.3.7)Removing the elements from Array using $pull operator:- We can use $pull operator either

1. To remove all instances of specified element.

2. To remove elements that match the given condition.

Syntax:

-------

{

$pull: {<array>: <value> | <condition> }

}

Example:-

> db.mca.insertOne({'name':'suku',age:[1,1,2,3,4,2,1]});

{

"acknowledged" : true,

"insertedId" : ObjectId("6184ac5ea3e09f155ac2f76d")

}

> db.mca.update({},{$pull:{age:1}});

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

> db.mca.find();

{ "\_id" : ObjectId("6184ac5ea3e09f155ac2f76d"), "name" : "suku", "age" : [ 2, 3, 4, 2 ] }

**6.4) Delete**:- There are 3 methods to delete the documents from collection

i)deleteOne() ii)deleteMany iii)remove.

**a) delete**:- To delete only one document that matches the query criteria.

> db.collection.deleteOne({query})

> db.mca.find();

{ "\_id" : ObjectId("6183ac2a66de04c1e59c9c97"), "name" : "suku", "age" : [ 777, 30, 30, 3, 4, 38, 41, 39, 40, 41 ] }

{ "\_id" : ObjectId("6183ac3966de04c1e59c9c98"), "name" : "suku1", "age" : [ 777, 11, 30, 30, 14, 38, 41, 39, 40, 41 ] }

{ "\_id" : ObjectId("6183ac4766de04c1e59c9c99"), "name" : "suku2", "age" : [ 777, 21, 22, 23, 24, 38, 41, 39, 40, 41 ] }

> db.mca.deleteOne({age:777});

{ "acknowledged" : true, "deletedCount" : 1 }

> db.mca.find();

{ "\_id" : ObjectId("6183ac3966de04c1e59c9c98"), "name" : "suku1", "age" : [ 777, 11, 30, 30, 14, 38, 41, 39, 40, 41 ] }

{ "\_id" : ObjectId("6183ac4766de04c1e59c9c99"), "name" : "suku2", "age" : [ 777, 21, 22, 23, 24, 38, 41, 39, 40, 41 ] }

**b) deleteMany**:- To delete all matched documents that matches query criteria.170

> db.collection.deleteMany({query})

> db.mca.find();

{ "\_id" : ObjectId("6183ac3966de04c1e59c9c98"), "name" : "suku1", "age" : [ 777, 11, 30, 30, 14, 38, 41, 39, 40, 41 ] }

{ "\_id" : ObjectId("6183ac4766de04c1e59c9c99"), "name" : "suku2", "age" : [ 777, 21, 22, 23, 24, 38, 41, 39, 40, 41 ] }

> db.mca.deleteMany({age:777});

{ "acknowledged" : true, "deletedCount" : 2 }

> db.mca.find();

>

**c) Remove**:- The remove command is used to delete the documents from collection.

Syntax:- db.collection\_name.remove(deletion criteria,1)

The second parameter is optional, it is either 0 or 1. Default value is 0. If it is 1 then only the first document which met the criterial will be removed.

If it is 0 then all documents which met the criteria will be removed.

Example:-

> db.student.find();

{ "\_id" : ObjectId("6175525a278d76fe7b787d1e"), "name" : "suku", "age" : 40 }

{ "\_id" : ObjectId("61755271278d76fe7b787d1f"), "name" : "sv", "age" : 36 }

{ "\_id" : ObjectId("61755911278d76fe7b787d20"), "name" : "suku", "age" : 37 }

{ "\_id" : ObjectId("61755926278d76fe7b787d21"), "name" : "veena", "age" : 37 }

> db.student.remove({name:'suku'});

WriteResult({ "nRemoved" : 2 })

> db.student.find();

{ "\_id" : ObjectId("61755271278d76fe7b787d1f"), "name" : "sv", "age" : 36 }

{ "\_id" : ObjectId("61755926278d76fe7b787d21"), "name" : "veena", "age" : 37 }

Example:2

> db.student.find();

{ "\_id" : ObjectId("61755271278d76fe7b787d1f"), "name" : "sv", "age" : 36 }

{ "\_id" : ObjectId("61755926278d76fe7b787d21"), "name" : "veena", "age" : 37 }

{ "\_id" : ObjectId("617559a2278d76fe7b787d22"), "name" : "veena", "age" : 40 }

> db.student.remove({name:'veena'},1);

WriteResult({ "nRemoved" : 1 })

> db.student.find();

{ "\_id" : ObjectId("61755271278d76fe7b787d1f"), "name" : "sv", "age" : 36 }

{ "\_id" : ObjectId("617559a2278d76fe7b787d22"), "name" : "veena", "age" : 40 }

7.Cursor:- In a collection there may be a chance of lakhs of documents. Whenever we are trying to retrieve data from database, if MongoDB server sends total data, there may be a chance of the following problems:

1. when main memory size is less than total date,Storage problem is occurred.

2. If MongoDb server is in remote machine then the Network traffic problem may be happened.

3. Performance problems

etc

To prevent these problems, most of the databases including MongoDB, uses cursor concept. The find() method / command always returns the ‘cursor’ object. Using cursor object , we get document either batch wise of one by one. Bydefault cursor object will provide documents in batch wise. The default batch size is 20.

We can change the batch default size DBQuery.shellBatchSize property.

Example:-

-----------

> DBQuery.shellBatchSize=3;

3

> db.mca.find();

{ "\_id" : ObjectId("6182147baaa527fd94d17fd1"), "SNO" : 1, "SNAME" : "x", "Marks" : 77 }

{ "\_id" : ObjectId("6182147baaa527fd94d17fd2"), "SNO" : 4, "SNAME" : "abc", "Marks" : 80 }

{ "\_id" : ObjectId("6182147baaa527fd94d17fd3"), "SNO" : 2, "SNAME" : "y", "Marks" : 78 }

**Type "it" for more**

**> it**

{ "\_id" : ObjectId("6182147baaa527fd94d17fd4"), "SNO" : 3, "SNAME" : "z", "Marks" : 79 }

>

**Methods of Cursor Object:**

**1.count():- It returns total no.of documents in cursor.**

**2.hasNext():-If cursor has next document then it returns true. Otherwise it returns false.**

**3.next():- It returns next document from cursor . It next document is not available, then it returns the error.**

**Example:-**

**--------------**

**> var c1=db.mca.find()**

**> while(c1.hasNext()){ print(c1.next());}**

**[object BSON]**

**[object BSON]**

**[object BSON]**

**[object BSON]**

**> var c1=db.mca.find()**

**> while(c1.hasNext()){ printjson(c1.next());}**

**{**

**"\_id" : ObjectId("6182147baaa527fd94d17fd1"),**

**"SNO" : 1,**

**"SNAME" : "x",**

**"Marks" : 77**

**}**

**{**

**"\_id" : ObjectId("6182147baaa527fd94d17fd2"),**

**"SNO" : 4,**

**"SNAME" : "abc",**

**"Marks" : 80**

**}**

**{**

**"\_id" : ObjectId("6182147baaa527fd94d17fd3"),**

**"SNO" : 2,**

**"SNAME" : "y",**

**"Marks" : 78**

**}**

**{**

**"\_id" : ObjectId("6182147baaa527fd94d17fd4"),**

**"SNO" : 3,**

**"SNAME" : "z",**

**"Marks" : 79**

**}**

> c1.forEach((a)=>{printjson(a)});

{

"\_id" : ObjectId("6182147baaa527fd94d17fd1"),

"SNO" : 1,

"SNAME" : "x",

"Marks" : 77

}

{

"\_id" : ObjectId("6182147baaa527fd94d17fd2"),

"SNO" : 4,

"SNAME" : "abc",

"Marks" : 80

}

{

"\_id" : ObjectId("6182147baaa527fd94d17fd3"),

"SNO" : 2,

"SNAME" : "y",

"Marks" : 78

}

{

"\_id" : ObjectId("6182147baaa527fd94d17fd4"),

"SNO" : 3,

"SNAME" : "z",

"Marks" : 79

}

**Cursor Helper Methods**:- To shape our result from cursor, we use following helper methods.

1.limit() 2. Skip() 3.sort()

1.limit():-

The default batch size is 20 documents. If you want to limit the documents from 20 documents, then we should use limit method.

Syntax:- db.collection-name.find().limit(n)

> db.mca.find().limit(2);

{ "\_id" : ObjectId("6182147baaa527fd94d17fd1"), "SNO" : 1, "SNAME" : "x", "Marks" : 77 }

{ "\_id" : ObjectId("6182147baaa527fd94d17fd2"), "SNO" : 4, "SNAME" : "abc", "Marks" : 80 }

Note:- The limit should be less than default batch size or custom batch size. If we specify greathe than default size or custom batch size, then this method does not has any impact.

b)$skip:- It skips n no.of documents from the result. It selects only rest of the documents.

Syntax:- db.collection-name.find().skip(n);

Where n is positive number or negative number.

If n should be positive then skipping is started from beginning.

Example:-

> db.mca.find();

{ "\_id" : ObjectId("6182147baaa527fd94d17fd1"), "SNO" : 1, "SNAME" : "x", "Marks" : 77 }

{ "\_id" : ObjectId("6182147baaa527fd94d17fd2"), "SNO" : 4, "SNAME" : "abc", "Marks" : 80 }

{ "\_id" : ObjectId("6182147baaa527fd94d17fd3"), "SNO" : 2, "SNAME" : "y", "Marks" : 78 }

{ "\_id" : ObjectId("6182147baaa527fd94d17fd4"), "SNO" : 3, "SNAME" : "z", "Marks" : 79 }

> db.mca.find().skip(2);

{ "\_id" : ObjectId("6182147baaa527fd94d17fd3"), "SNO" : 2, "SNAME" : "y", "Marks" : 78 }

{ "\_id" : ObjectId("6182147baaa527fd94d17fd4"), "SNO" : 3, "SNAME" : "z", "Marks" : 79 }

c)sort():-We can use sort() method to sort documents based on value of a particular field.

Syntax:- sort({field: <value>})

Sort({field:<value>, field1:<value>,…etc});

Where value is either 1 or -1 . 1 means ascending order and -1 means descending order.

Example Query:- select 2,3 documents from any collection.

> db.mca.find();

{ "\_id" : ObjectId("6182147baaa527fd94d17fd1"), "SNO" : 1, "SNAME" : "x", "Marks" : 77 }

{ "\_id" : ObjectId("6182147baaa527fd94d17fd2"), "SNO" : 4, "SNAME" : "abc", "Marks" : 80 }

{ "\_id" : ObjectId("6182147baaa527fd94d17fd3"), "SNO" : 2, "SNAME" : "y", "Marks" : 78 }

{ "\_id" : ObjectId("6182147baaa527fd94d17fd4"), "SNO" : 3, "SNAME" : "z", "Marks" : 79 }

>

> db.mca.find().skip(1).limit(2);

{ "\_id" : ObjectId("6182147baaa527fd94d17fd2"), "SNO" : 4, "SNAME" : "abc", "Marks" : 80 }

{ "\_id" : ObjectId("6182147baaa527fd94d17fd3"), "SNO" : 2, "SNAME" : "y", "Marks" : 78 }

> db.mca.find().skip(1).limit(2).sort({Marks:1});

{ "\_id" : ObjectId("6182147baaa527fd94d17fd3"), "SNO" : 2, "SNAME" : "y", "Marks" : 78 }

{ "\_id" : ObjectId("6182147baaa527fd94d17fd4"), "SNO" : 3, "SNAME" : "z", "Marks" : 79 }

8.Projection:- We can get documents with only required fields instead of all fields. This is called projection.

Syntax:- db.collectioname.find({ filter},{projection fields});

Fied syntax:-

------------------

Field:0 or 1

0 means field does not include in the result.

1 means field will be included in the result. If we are not taking any field in the projected list, **bydefault that field will be excluded. ie default value is 0**.

Note: If we are providing projection list, compulsory we should provide filter object also, atleast empty java script object{}. i.e without providing first argument, we cannot talk about second argument.

Example:-

------------

> db.mca.find();

{ "\_id" : ObjectId("6182147baaa527fd94d17fd1"), "SNO" : 1, "SNAME" : "x", "Marks" : 77 }

{ "\_id" : ObjectId("6182147baaa527fd94d17fd2"), "SNO" : 4, "SNAME" : "abc", "Marks" : 80 }

{ "\_id" : ObjectId("6182147baaa527fd94d17fd3"), "SNO" : 2, "SNAME" : "y", "Marks" : 78 }

{ "\_id" : ObjectId("6182147baaa527fd94d17fd4"), "SNO" : 3, "SNAME" : "z", "Marks" : 79 }

> db.mca.find(,{\_id:0,sno:1});

uncaught exception: SyntaxError: expected expression, got ',' :

@(shell):1:12

> db.mca.find({\_id:0,sno:1});

> db.mca.find({},{\_id:0,SNO:1});

{ "SNO" : 1 }

{ "SNO" : 4 }

{ "SNO" : 2 }

{ "SNO" : 3 }

> db.mca.find({},{\_id:0,SNO:1,SNAME:1});

{ "SNO" : 1, "SNAME" : "x" }

{ "SNO" : 4, "SNAME" : "abc" }

{ "SNO" : 2, "SNAME" : "y" }

{ "SNO" : 3, "SNAME" : "z" }

> db.mca.find({},{\_id:0,SNO:1,Marks:1});

{ "SNO" : 1, "Marks" : 77 }

{ "SNO" : 4, "Marks" : 80 }

{ "SNO" : 2, "Marks" : 78 }

{ "SNO" : 3, "Marks" : 79 }

> db.mca.find(SNO:1},{\_id:0,SNO:1,Marks:1});

{ "SNO" : 1, "Marks" : 77 }

8.1)Projection of Nested Documents:- In projection field list, The field name of inner document is accessed using (.) operator and outerfieldname and innerfield name must be written inside a double quotes.

EX:- “outer field name.inner fieldname”: 1 or 0.

Example:-

> db.mca.find();

{ "\_id" : ObjectId("6182574e48c263ddf7c93e9f"), "name" : "suku", "addr" : { "street" : "rajka street", "town" : "nlr" } }

{ "\_id" : ObjectId("6182575d48c263ddf7c93ea0"), "name" : "suku1", "addr" : { "street" : "rajka street1", "town" : "nlr1" } }

> db.mca.find({},{\_id:0,"addr.street":1});

{ "addr" : { "street" : "rajka street" } }

{ "addr" : { "street" : "rajka street1" } }

> db.mca.find({name:'suku'},{\_id:0,"addr.town":1});

{ "addr" : { "town" : "nlr" } }

8.2)Projection of Arrays:-

Example:- In the below example , we are going to project all elements of Array.

> db.mca.find();

{ "\_id" : ObjectId("6182597a48c263ddf7c93ea1"), "name" : "suku", "age" : [ 1, 2, 3, 4, 5 ] }

{ "\_id" : ObjectId("6182598748c263ddf7c93ea2"), "name" : "suku1", "age" : [ 11, 12, 13, 14, 15 ] }

{ "\_id" : ObjectId("6182599548c263ddf7c93ea3"), "name" : "suku2", "age" : [ 21, 22, 23, 24, 25 ] }

> db.mca.find({},{\_id:0,age:1});

{ "age" : [ 1, 2, 3, 4, 5 ] }

{ "age" : [ 11, 12, 13, 14, 15 ] }

{ "age" : [ 21, 22, 23, 24, 25 ] }

We can control projection of elements of array using following operators.

1. $

2. $elemMatch

3. $slice

8.2.1) $:-We can use $ operator to project first element in an array that matches query condition.

Syntax: db.collection.find({<array>:<condition>,...},{"<array>.$":1})

**Note: If there is no query condition or if query condition won't include array then we cannot use $ operator, otherwise we will get error.**

**: $ operator selects only one element which is first matched element based on query condition.**

Example:-

> db.mca.find();

{ "\_id" : ObjectId("6182597a48c263ddf7c93ea1"), "name" : "suku", "age" : [ 1, 2, 3, 4, 5 ] }

{ "\_id" : ObjectId("6182598748c263ddf7c93ea2"), "name" : "suku1", "age" : [ 11, 12, 13, 14, 15 ] }

{ "\_id" : ObjectId("6182599548c263ddf7c93ea3"), "name" : "suku2", "age" : [ 21, 22, 23, 24, 25 ] }

> db.mca.find({age:{$gt:12}},{"age.$":1});

{ "\_id" : ObjectId("6182598748c263ddf7c93ea2"), "age" : [ 13 ] }

{ "\_id" : ObjectId("6182599548c263ddf7c93ea3"), "age" : [ 21 ] }

8.2.2) 2. $elemMatch operator:

-----------------------

1. selects only one element

2. which is matched element where condition is specified by

$elemMatch explicitly.

It never considers query condition.

We can use $elemMatch to project first element in the array that matches specified $elemMatch condition.

Example:-

> db.mca.find();

{ "\_id" : ObjectId("6182597a48c263ddf7c93ea1"), "name" : "suku", "age" : [ 1, 2, 3, 4, 5 ] }

{ "\_id" : ObjectId("6182598748c263ddf7c93ea2"), "name" : "suku1", "age" : [ 11, 12, 13, 14, 15 ] }

{ "\_id" : ObjectId("6182599548c263ddf7c93ea3"), "name" : "suku2", "age" : [ 21, 22, 23, 24, 25 ] }

> db.mca.find({},{\_id:0,'name':1,age:{$elemMatch:{$gt:13}}});

{ "name" : "suku" }

{ "name" : "suku1", "age" : [ 14 ] }

{ "name" : "suku2", "age" : [ 21 ] }

8.2.3) $slice operator:

-------------------

By using $slice operator we can select required number of elements in

the array.

Syntax-1:

---------

db.collection.find({query},{<array>:{$slice: n}})

n-->number of elements to be selected.

Specify a positive number n to return the first n elements.

Specify a negative number n to return the last n elements.If n is greater than number of elements in the array then all elements will be selected.

Examples:-

> db.mca.find();

{ "\_id" : ObjectId("6182597a48c263ddf7c93ea1"), "name" : "suku", "age" : [ 1, 2, 3, 4, 5 ] }

{ "\_id" : ObjectId("6182598748c263ddf7c93ea2"), "name" : "suku1", "age" : [ 11, 12, 13, 14, 15 ] }

{ "\_id" : ObjectId("6182599548c263ddf7c93ea3"), "name" : "suku2", "age" : [ 21, 22, 23, 24, 25 ] }

> db.mca.find({},{\_id:0,name:1,age:{$slice:3}});

{ "name" : "suku", "age" : [ 1, 2, 3 ] }

{ "name" : "suku1", "age" : [ 11, 12, 13 ] }

{ "name" : "suku2", "age" : [ 21, 22, 23 ] }

> db.mca.find({},{\_id:0,name:1,age:{$slice:-3}});

{ "name" : "suku", "age" : [ 3, 4, 5 ] }

{ "name" : "suku1", "age" : [ 13, 14, 15 ] }

{ "name" : "suku2", "age" : [ 23, 24, 25 ] }

>

Syntax:-2

------------

db.collection.find({query},{<array>:{$slice: [n1,n2]}})

skip n1 number of elements and then select n2 number of elements.

n1--->number to skip

n2--->number to return

> db.mca.find();

{ "\_id" : ObjectId("6182597a48c263ddf7c93ea1"), "name" : "suku", "age" : [ 1, 2, 3, 4, 5 ] }

{ "\_id" : ObjectId("6182598748c263ddf7c93ea2"), "name" : "suku1", "age" : [ 11, 12, 13, 14, 15 ] }

{ "\_id" : ObjectId("6182599548c263ddf7c93ea3"), "name" : "suku2", "age" : [ 21, 22, 23, 24, 25 ] }

> db.mca.find({},{\_id:0,name:1,age:{$slice:[1,3]}});

{ "name" : "suku", "age" : [ 2, 3, 4 ] }

{ "name" : "suku1", "age" : [ 12, 13, 14 ] }

{ "name" : "suku2", "age" : [ 22, 23, 24 ] }

9.MongoDB Utilities/Tools:- The following are mongodb utitilites.

a)mongoimport

b)mongoexport

c)mongodump

d)mongorestore.

..etc.

These are separate applications. These commands should not be executed from mongoDB shell. These commands are executed from command prompt. These are used to manage the data in mongoDB database.

These tools are not available along with mongo db server by default. We have to explicitly install them.

Steps to install these tools.

Step1:- visit following url.

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

Step2. Click the download button.

Step3.we will get this file “ mongodb-database-tools-windows-x86\_64-100.5.1”. This is zip file.

Step4. Extract this zip file and copy all the file. Paste them in mongodb ‘ bin’ folder.

9.1)mongoexport:- we use this tool to export data from collection to files. The data will be stored as json format in files.

Syntax:-

Mongoexport –d databasename -c collection name -o name of the file

Example:-

C:\Program Files\MongoDB\Server\5.0\bin>mongoexport -d Library -c mca -o "E:\abc.txt"

2021-11-05T10:28:16.061+0530 connected to: mongodb://localhost/

2021-11-05T10:28:16.580+0530 exported 1 record

9.2)mongodump:- we can take the backup of the database using mongodump command.

(or) to create dump from db, we use this command.

Using this command, we dump all the databases,particular database ,all collections and particular collections.

9.2.1)create dump for all databases.:-

Step1: Go to bin folder in mongoDb server.

Step2: run the following command.

>mongodump

Now , folder named **‘dump’** will be created. This folder contains all data from databases.

In this folder, for every collection,two files will be created. One file type is .json file which contains meta data . Another file type is .Bson type which contains actual documents. The data which is in .Bson file is not in readable format.

We ‘bsondump’ tool is used to convert bson data to json format .

Syntax1:- bsondump filename.bson

Here bsondump tool converts data from bson to json data and write that to console.

Syntax2:- bsondump --outFile=new filename filename.bson

Here bsonddump tool converts data from bson to json data and write that to specified file instead of console.

9.2.2) create dump for particular database:-

Syntax:- mongodump --db databasename.

9.2.3) create dump for single collection:-

Syntax:- mongodump –db databasename –c collectionName

9.3)mongorestore:- This command is used to restore all databases,particulat database, all collections and particular collection in ‘mongoDB physical db’.

9.3.1) Restore all databases:-

Step1:- go to the directory which contains the ‘dump’ folder.

Step2:- run the following command.

>mongorestore.

9.3.2) Restore Particular Database:-

Syntax:- mongorestore --db databasename absolute path name of databasename.

Note :- The database which is in ‘dump’ folder. Its absolute path name is written in syntax.

9.3.3) Restore a single collection:-

Syntax:- mongorestore –db databasename –collection collectionname absolute path name of collection name.

Note:- The collection which is in ‘dump’ folder. Its absolute path name is written in syntax.

**10.GUI Tools For MongoDB Operations**:- Upto this we performed the database operations by using mongo shell.

The advantage of using shell is we have to do everything so that

learning opportunity is more.

But in real time usage of shell is not recommended because of the

following reasons:

1. Auto completion is not available. We have to type complete

command.

2. While writing complex queries, more error prone.188

3. No help tips

4. Readability is not up to the mark.

5. No coloring.

6. Operations will become very complex

7. Not that much convenient to use

etc

To overcome these problems we have to use GUI based tools like

1. Robo 3T

2. Studio 3T

3. Compass

4. NoSQL Manager

etc

10.1) Robo 3T: It is freeware and lightweight GUI tool for MongoDB operations. website: robomongo.org

Two tools: Robo 3T and Studio 3T

Robo 3T is freeware and Studio 3T is licensed.

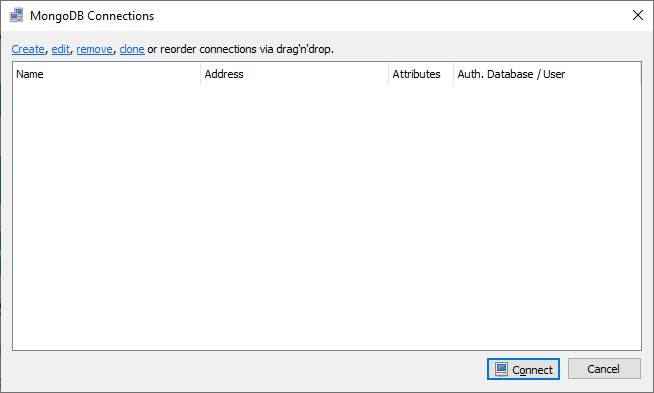
Download: studio-3t-robo-3t-windows-double-pack.zip (or).exe .

Install it.

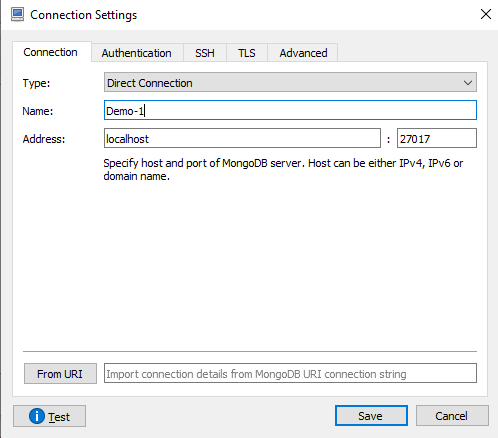
A)Steps to create Database using Robo3T:-

----------------------------------------------------

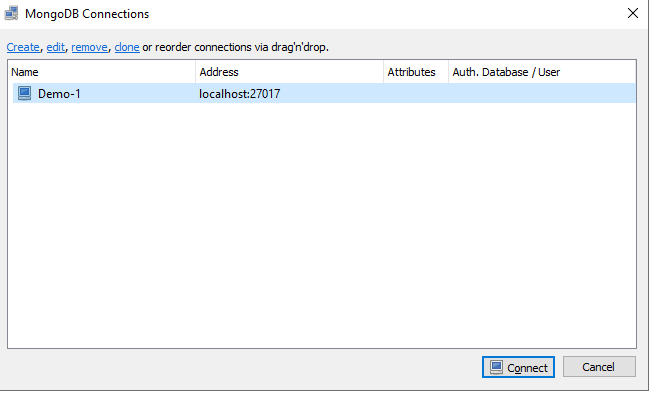
Step1: create a DB connection.



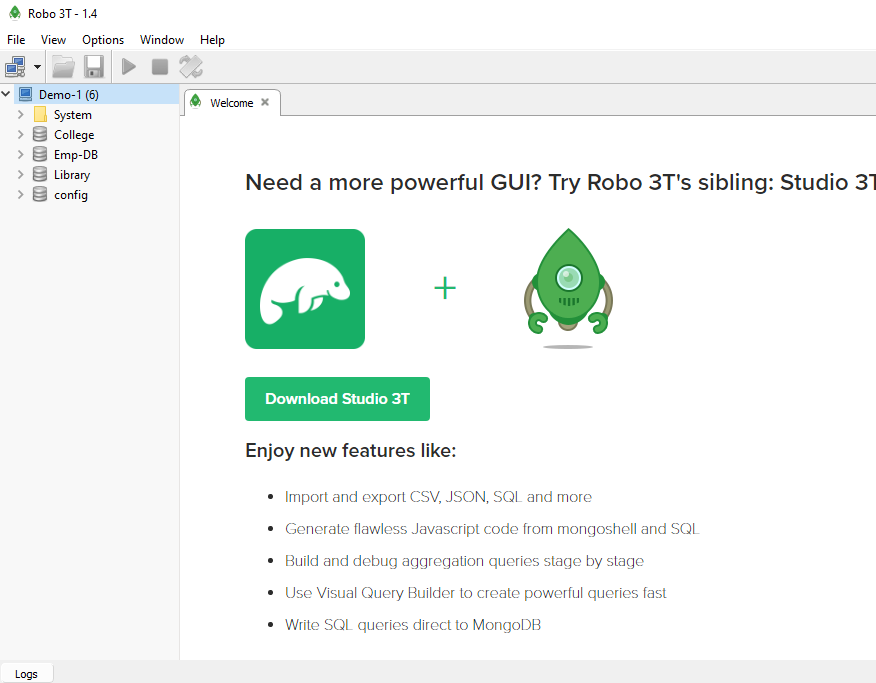
After clicking the ‘create’ we see below screen.



After entering Name of connection, click the save button. After clicking the save button, connection is created.

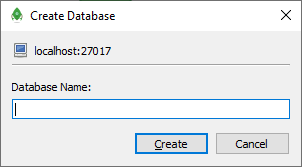


Click the connect button. Now you are going to connect the mongodb server.



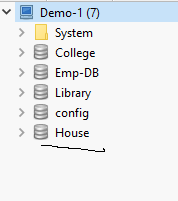
Step2:- Select the Demo-1,Right click the mouse button. You will see the drop down list. From the drop down list ,select the create database option.

After selecting ,we will sell the below window.

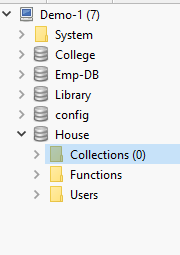


Type the database and press the create button.

Now Db will be created.

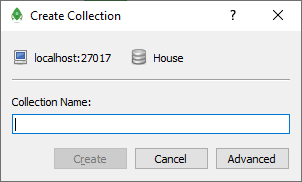


B)Steps to create a Collection using Robot-3T:-



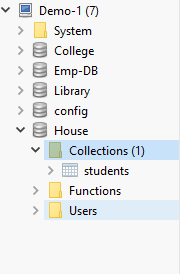
Step1:- Select the Collection and right click the mouse. Now we will see the list. From the list select ‘Create Collection’ .

After selecting, we will see the below window.



Type the collection name(student) and press the create button.

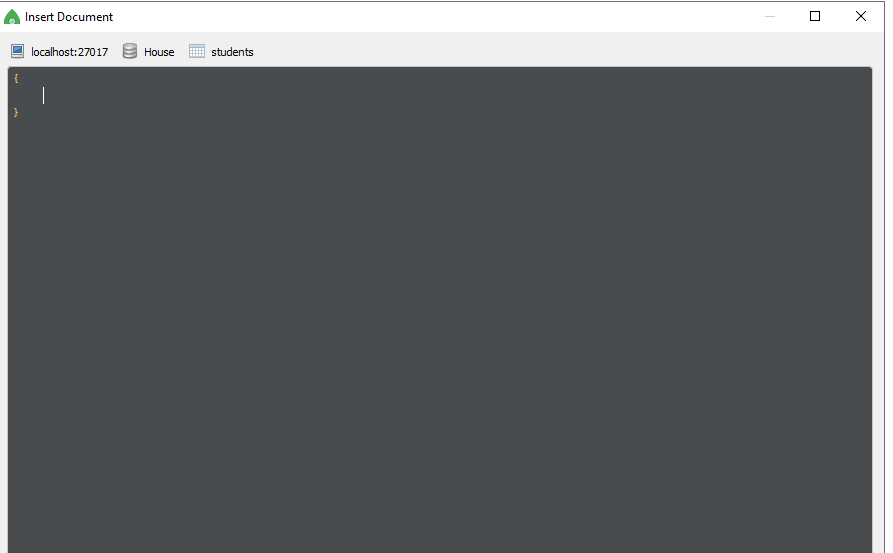
Now Collection will be created under the Collection folder.



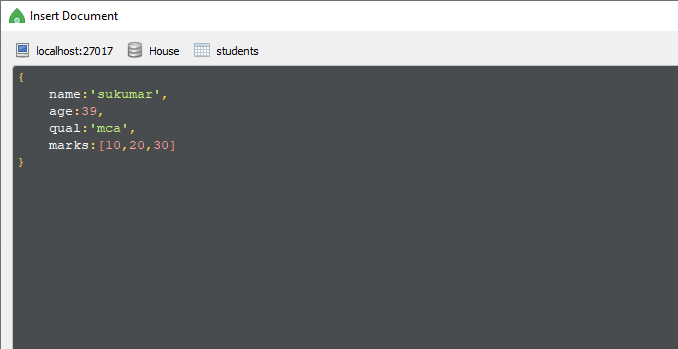
C)Steps to create Document in Students Collection:-

------------------------------------------------------------------

Step1. Select ‘students’ and right click the mouse. Now you will see the list . From the list you select the ‘create document’.

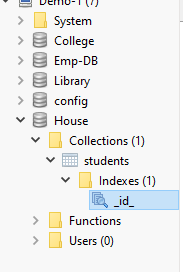


Step2: type the document.



Press the ‘save’ button.

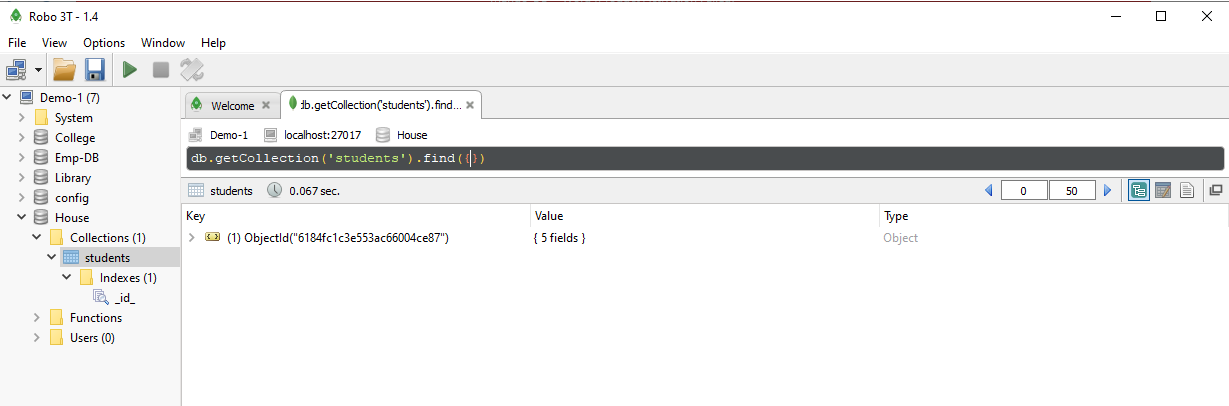
Now one document is created.



d)steps to see the documents in collection:-

-------------------------------------------------------

After clicking the students, we see the documents in ‘students’ collection.



Note:- The document can be viewed as tree,table or text . To see like that, we use icons which is top right corner of sub window.

Note:- For downloading and installing the other gui tools, visit the ‘Durga sir ‘ material.

11.Indexing:-

Problem:- Without indexes, server will scan all documents present in the collection to find matched documents. This is called Collection Scan(COLLSCAN). Example:-

> db.mca.find();

{ "\_id" : ObjectId("6184be5fa3e09f155ac2f76e"), "name" : "suku", "age" : 39 }

{ "\_id" : ObjectId("6184be68a3e09f155ac2f76f"), "name" : "suku1", "age" : 40 }

{ "\_id" : ObjectId("6184be6fa3e09f155ac2f770"), "name" : "suku2", "age" : 41 }

{ "\_id" : ObjectId("6184df1c9b6920f4b7c6886e"), "name" : "rock", "age" : 45 }

{ "\_id" : ObjectId("6184df559b6920f4b7c6886f"), "name" : "rock1", "age" : 54 }

> db.mca.find({name:'rock'});

{ "\_id" : ObjectId("6184df1c9b6920f4b7c6886e"), "name" : "rock", "age" : 45 }

> db.mca.find({name:'rock'}).explain("executionStats");

{

"explainVersion" : "1",

"queryPlanner" : {

"namespace" : "College.mca",

"indexFilterSet" : false,

"parsedQuery" : {

"name" : {

"$eq" : "rock"

}

},

"maxIndexedOrSolutionsReached" : false,

"maxIndexedAndSolutionsReached" : false,

"maxScansToExplodeReached" : false,

"winningPlan" : {

"stage" : "COLLSCAN",

"filter" : {

"name" : {

"$eq" : "rock"

}

},

"direction" : "forward"

},

"rejectedPlans" : [ ]

},

"executionStats" : {

"executionSuccess" : true,

"nReturned" : 1,

"executionTimeMillis" : 100,

"totalKeysExamined" : 0,

"totalDocsExamined" : 5,

"executionStages" : {

"stage" : "COLLSCAN",

"filter" : {

"name" : {

"$eq" : "rock"

}

},

"nReturned" : 1,

"executionTimeMillisEstimate" : 0,

"works" : 7,

"advanced" : 1,

"needTime" : 5,

"needYield" : 0,

"saveState" : 1,

"restoreState" : 1,

"isEOF" : 1,

"direction" : "forward",

"docsExamined" : 5

}

},

"command" : {

"find" : "mca",

"filter" : {

"name" : "rock"

},

"$db" : "College"

},

"serverInfo" : {

"host" : "DESKTOP-NKQ12R1",

"port" : 27017,

"version" : "5.0.3",

"gitVersion" : "657fea5a61a74d7a79df7aff8e4bcf0bc742b748"

},

"serverParameters" : {

"internalQueryFacetBufferSizeBytes" : 104857600,

"internalQueryFacetMaxOutputDocSizeBytes" : 104857600,

"internalLookupStageIntermediateDocumentMaxSizeBytes" : 104857600,

"internalDocumentSourceGroupMaxMemoryBytes" : 104857600,

"internalQueryMaxBlockingSortMemoryUsageBytes" : 104857600,

"internalQueryProhibitBlockingMergeOnMongoS" : 0,

"internalQueryMaxAddToSetBytes" : 104857600,

"internalDocumentSourceSetWindowFieldsMaxMemoryBytes" : 104857600

},

"ok" : 1

}

Note:- mongodb server scans all documents in ‘mca’ collection for 1 record.

Observe following field in .explain() output.

"totalDocsExamined" : 5,

For selecting document, Mongodb server used “CollScan”.

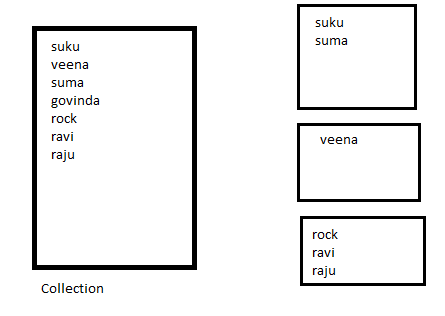
Here Query performance might be poor.

Let us assume, if collection contains 1 lack document,then Mongodb server has to scan 1 lack document for result. This leads to query performance problem. To solve these problems , we should use indexing concept.

**Indexing concept is very helpful for find,update and delete queries.The main objective of indexing is to improve performance**.whenever we are defining index, MongoDB Server will store values of indexed field in B-Tree Data structure in specified sorting order.To find matched documents, we are not required to scan all

documents,server can identify matched documents directly based Index Scan(IXSCAN). As the number of documents to be scan, is reduces and hence performance will be improved.

Example:-



Mongodb server places values some where in specified order . when ‘suku’ is going to be searched, server will search first box. It will not search remaining two boxes. Hence scanning the no.of document s will be reduced.

**Syntax to create Index**:

db.collection.createIndex({field: 1|-1})

1 means Ascending order

-1 means Descending order

**Syntax to Drop the index**:

db.collection.dropIndex({field:1|-1})

**syntax to display list of indexes:**

db.collection.getIndexes()

**Syntax to create Index on multiple fields:**

db.collection.createIndex({field1: 1, field2: -1})

The order of fields is important.

Examples:-

--------------

> db.mca.find();

{ "\_id" : ObjectId("6184be5fa3e09f155ac2f76e"), "name" : "suku", "age" : 39 }

{ "\_id" : ObjectId("6184be68a3e09f155ac2f76f"), "name" : "suku1", "age" : 40 }

{ "\_id" : ObjectId("6184be6fa3e09f155ac2f770"), "name" : "suku2", "age" : 41 }

{ "\_id" : ObjectId("6184df1c9b6920f4b7c6886e"), "name" : "rock", "age" : 45 }

{ "\_id" : ObjectId("6184df559b6920f4b7c6886f"), "name" : "rock1", "age" : 54 }

> db.mca.createIndex({name:1});

{

"numIndexesBefore" : 1,

"numIndexesAfter" : 2,

"createdCollectionAutomatically" : false,

"ok" : 1

}

> db.mca.showIndexes();

uncaught exception: TypeError: db.mca.showIndexes is not a function :

@(shell):1:1

> db.mca.getIndexes();

[

{

"v" : 2,

"key" : {

"\_id" : 1

},

"name" : "\_id\_"

},

{

"v" : 2,

"key" : {

"name" : 1

},

"name" : "name\_1"

}

]

> db.mca.createIndex({age:1});

{

"numIndexesBefore" : 2,

"numIndexesAfter" : 3,

"createdCollectionAutomatically" : false,

"ok" : 1

}

> db.mca.getIndexes()

[

{

"v" : 2,

"key" : {

"\_id" : 1

},

"name" : "\_id\_"

},

{

"v" : 2,

"key" : {

"name" : 1

},

"name" : "name\_1"

},

{

"v" : 2,

"key" : {

"age" : 1

},

"name" : "age\_1"

}

]

> db.mca.dropIndex({age:1});

{ "nIndexesWas" : 3, "ok" : 1 }

> db.mca.getIndexes();

[

{

"v" : 2,

"key" : {

"\_id" : 1

},

"name" : "\_id\_"

},

{

"v" : 2,

"key" : {

"name" : 1

},

"name" : "name\_1"

}

]

Practically shown the performance :-

----------------------------------------------

> db.mca.find();

{ "\_id" : ObjectId("6184be5fa3e09f155ac2f76e"), "name" : "suku", "age" : 39 }

{ "\_id" : ObjectId("6184be68a3e09f155ac2f76f"), "name" : "suku1", "age" : 40 }

{ "\_id" : ObjectId("6184be6fa3e09f155ac2f770"), "name" : "suku2", "age" : 41 }

{ "\_id" : ObjectId("6184df1c9b6920f4b7c6886e"), "name" : "rock", "age" : 45 }

{ "\_id" : ObjectId("6184df559b6920f4b7c6886f"), "name" : "rock1", "age" : 54 }

> db.mca.find({name:'rock1'}).explain("executionStats");

{

"explainVersion" : "1",

"queryPlanner" : {

"namespace" : "College.mca",

"indexFilterSet" : false,

"parsedQuery" : {

"name" : {

"$eq" : "rock1"

}

},

"maxIndexedOrSolutionsReached" : false,

"maxIndexedAndSolutionsReached" : false,

"maxScansToExplodeReached" : false,

"winningPlan" : {

"stage" : "FETCH",

"inputStage" : {

"stage" : "IXSCAN",

"keyPattern" : {

"name" : 1

},

"indexName" : "name\_1",

"isMultiKey" : false,

"multiKeyPaths" : {

"name" : [ ]

},

"isUnique" : false,

"isSparse" : false,

"isPartial" : false,

"indexVersion" : 2,

"direction" : "forward",

"indexBounds" : {

"name" : [

"[\"rock1\", \"rock1\"]"

]

}

}

},

"rejectedPlans" : [ ]

},

"executionStats" : {

"executionSuccess" : true,

"nReturned" : 1,

"executionTimeMillis" : 60,

"totalKeysExamined" : 1,

"totalDocsExamined" : 1,

"executionStages" : {

"stage" : "FETCH",

"nReturned" : 1,

"executionTimeMillisEstimate" : 11,

"works" : 2,

"advanced" : 1,

"needTime" : 0,

"needYield" : 0,

"saveState" : 1,

"restoreState" : 1,

"isEOF" : 1,

"docsExamined" : 1,

"alreadyHasObj" : 0,

"inputStage" : {

"stage" : "IXSCAN",

"nReturned" : 1,

"executionTimeMillisEstimate" : 11,

"works" : 2,

"advanced" : 1,

"needTime" : 0,

"needYield" : 0,

"saveState" : 1,

"restoreState" : 1,

"isEOF" : 1,

"keyPattern" : {

"name" : 1

},

"indexName" : "name\_1",

"isMultiKey" : false,

"multiKeyPaths" : {

"name" : [ ]

},

"isUnique" : false,

"isSparse" : false,

"isPartial" : false,

"indexVersion" : 2,

"direction" : "forward",

"indexBounds" : {

"name" : [

"[\"rock1\", \"rock1\"]"

]

},

"keysExamined" : 1,

"seeks" : 1,

"dupsTested" : 0,

"dupsDropped" : 0

}

}

},

"command" : {

"find" : "mca",

"filter" : {

"name" : "rock1"

},

"$db" : "College"

},

"serverInfo" : {

"host" : "DESKTOP-NKQ12R1",

"port" : 27017,

"version" : "5.0.3",

"gitVersion" : "657fea5a61a74d7a79df7aff8e4bcf0bc742b748"

},

"serverParameters" : {

"internalQueryFacetBufferSizeBytes" : 104857600,

"internalQueryFacetMaxOutputDocSizeBytes" : 104857600,

"internalLookupStageIntermediateDocumentMaxSizeBytes" : 104857600,

"internalDocumentSourceGroupMaxMemoryBytes" : 104857600,

"internalQueryMaxBlockingSortMemoryUsageBytes" : 104857600,

"internalQueryProhibitBlockingMergeOnMongoS" : 0,

"internalQueryMaxAddToSetBytes" : 104857600,

"internalDocumentSourceSetWindowFieldsMaxMemoryBytes" : 104857600

},

"ok" : 1

}

Note:- MongoDb server scans only 1 document to select the result.

"docsExamined" : 1.

With out indexing on Collection, same query scanned 5 docments. But now it scanned only one document.

Note:-1. For small database ,Don’t create a indexes. For small databases ,Indexes leads to create more performance problem.

2.Dont create index on all fields of document. Create index on frequently used field in selection criteria.

EX:- IRCTC , we search available trains based on train number or train name. Therefore create index on train no or train-name field.

12.Aggeregate FrameWorks:-

Syntax:- db.collectio-name.aggregate{$group:{\_id:null(or)fieldname, text:{$accumalator- operator:”$fieldname”}}}

1. The first parameter in $group stage is always \_id. We should use \_id to specify field based on which we have to perform grouping of documents.

Ex:- $group:{\_id:rock,

1. If we want to process all records then we have to provide null value to \_id field.

Ex:- $group:{\_id:null,

1. The text will be displayed as it is on console.
2. The accumulator operators are $sum,$avg,$max,$min,…etc.

Examples:-

> db.mca.find();

{ "\_id" : ObjectId("6184be5fa3e09f155ac2f76e"), "name" : "suku", "age" : 39 }

{ "\_id" : ObjectId("6184be68a3e09f155ac2f76f"), "name" : "suku1", "age" : 40 }

{ "\_id" : ObjectId("6184be6fa3e09f155ac2f770"), "name" : "suku2", "age" : 41 }

{ "\_id" : ObjectId("6184df1c9b6920f4b7c6886e"), "name" : "rock", "age" : 45 }

{ "\_id" : ObjectId("6184df559b6920f4b7c6886f"), "name" : "rock1", "age" : 54 }

> db.mca.aggregate([{$group:{\_id:null, TotalAge:{$sum:'$age'}}}]);

{ "\_id" : null, "TotalAge" : 219 }

> db.mca.aggregate([{$group:{\_id:null, MaxAge:{$max:'$age'}}}]);

{ "\_id" : null, "MaxAge" : 54 }

> db.mca.aggregate([{$group:{\_id:null, MinAge:{$min:'$age'}}}]);

{ "\_id" : null, "MinAge" : 39 }

> db.mca.insertOne({'name':'suku',age:55});

{

"acknowledged" : true,

"insertedId" : ObjectId("61855fc34f3f45e941c81b30")

}

> db.mca.find();

{ "\_id" : ObjectId("6184be5fa3e09f155ac2f76e"), "name" : "suku", "age" : 39 }

{ "\_id" : ObjectId("6184be68a3e09f155ac2f76f"), "name" : "suku1", "age" : 40 }

{ "\_id" : ObjectId("6184be6fa3e09f155ac2f770"), "name" : "suku2", "age" : 41 }

{ "\_id" : ObjectId("6184df1c9b6920f4b7c6886e"), "name" : "rock", "age" : 45 }

{ "\_id" : ObjectId("6184df559b6920f4b7c6886f"), "name" : "rock1", "age" : 54 }

{ "\_id" : ObjectId("61855fc34f3f45e941c81b30"), "name" : "suku", "age" : 55 }

> db.mca.aggregate([{$group:{\_id:'$name', MinAge:{$min:'$age'}}}]);

{ "\_id" : "suku1", "MinAge" : 40 }

{ "\_id" : "rock1", "MinAge" : 54 }

{ "\_id" : "rock", "MinAge" : 45 }

{ "\_id" : "suku", "MinAge" : 39 }

{ "\_id" : "suku2", "MinAge" : 41 }

> db.mca.aggregate([{$group:{\_id:'$name', TotalAge:{$sum:'$age'}}}]);

{ "\_id" : "suku", "TotalAge" : 94 }

{ "\_id" : "suku2", "TotalAge" : 41 }

{ "\_id" : "rock1", "TotalAge" : 54 }

{ "\_id" : "rock", "TotalAge" : 45 }

{ "\_id" : "suku1", "TotalAge" : 40 }

>

12.1)Aggregate Pipeline:- We can define multiple stages in the aggregation and all these stages will form pipeline, which is known as aggregation pipeline.

db.collection.aggregate([

{stage-1},

{stage-2},

{stage-3},

{stage-4},

{stage-5}

...

])

The stages are executed one by one. One stage output will becomes an input to next stage. The collection becomes an input to stage-1. The final produced doucuments in required format.

There are several predefined stages like $group,$sort,$project,$match,$limit,…etc.

a)$sort:- It sorts all input documents in specified order and return them to next stage or console.

The $sort stage has the following prototype form:

{ $sort: { <field1>: <sort order>, <field2>: <sort order> ... } }

The <sort order> can be either 1 or -1.

1 --->Ascending Order

-1 ---> Descending Order

Example:-

> db.mca.find();

{ "\_id" : ObjectId("6185e323895a7fdfd4de05a7"), "name" : "suku", "age" : 37 }

{ "\_id" : ObjectId("6185e32a895a7fdfd4de05a8"), "name" : "suku2", "age" : 39 }

{ "\_id" : ObjectId("6185e338895a7fdfd4de05a9"), "name" : "suku3", "age" : 40 }

{ "\_id" : ObjectId("6185e340895a7fdfd4de05aa"), "name" : "suku", "age" : 41 }

> db.mca.aggregate($group:{\_id:"$name",TotalSalary:{$sum:"$age"}},

... {$sort:{TotalSalary:-1}});

uncaught exception: SyntaxError: missing ) after argument list :

@(shell):1:23

> db.mca.aggregate([$group:{\_id:"$name",TotalSalary:{$sum:"$age"}}, {$sort:{TotalSalary:-1}}]);

uncaught exception: SyntaxError: missing ] after element list :

@(shell):1:24

> db.mca.aggregate([{$group:{\_id:"$name",TotalSalary:{$sum:"$age"}}}, {$sort:{TotalSalary:-1}}]);

{ "\_id" : "suku", "TotalSalary" : 78 }

{ "\_id" : "suku3", "TotalSalary" : 40 }

{ "\_id" : "suku2", "TotalSalary" : 39 }

b)$project:- By using this $project stage, we can restrict documents with our required fields only.(i.e) we can add new field or delete exiting field from documents.

Syntax:

{ $project: { field:0|1 } }

0 or false --->To exclude the field

1 or true --->To include the field

Example:

------------

> db.mca.aggregate([{$group:{\_id:"$name",TotalSalary:{$sum:"$age"}}}, {$sort:{TotalSalary:-1}}]);

{ "\_id" : "suku", "TotalSalary" : 78 }

{ "\_id" : "suku3", "TotalSalary" : 40 }

{ "\_id" : "suku2", "TotalSalary" : 39 }

> db.mca.aggregate([{$group:{\_id:"$name",TotalSalary:{$sum:"$age"}}}, {$sort:{TotalSalary:-1}},{$project:{\_id:0}}]);

{ "TotalSalary" : 78 }

{ "TotalSalary" : 40 }

{ "TotalSalary" : 39 }

> db.mca.aggregate([{$group:{\_id:"$name",TotalSalary:{$sum:"$age"}}}, {$sort:{TotalSalary:-1}},{$project:{\_id:0,EmpName:"$\_id",TotalSalary:1}}]);

{ "TotalSalary" : 78, "EmpName" : "suku" }

{ "TotalSalary" : 40, "EmpName" : "suku3" }

{ "TotalSalary" : 39, "EmpName" : "suku2" }

>

c)$match:- To filter the documents from input , we use $match stage.

Syntax:- { $match: { <query> } }

Example:-

> db.mca.aggregate([{$group:{\_id:"$name",TotalSalary:{$sum:"$age"}}}, {$sort:{TotalSalary:-1}},{$project:{\_id:0,EmpName:"$\_id",TotalSalary:1}}]);

{ "TotalSalary" : 78, "EmpName" : "suku" }

{ "TotalSalary" : 40, "EmpName" : "suku3" }

{ "TotalSalary" : 39, "EmpName" : "suku2" }

> db.mca.aggregate([{$group:{\_id:"$name",TotalSalary:{$sum:"$age"}}}, {$sort:{TotalSalary:-1}},{$project:{\_id:0,EmpName:"$\_id",TotalSalary:1}},{$match:{TotalSalary:{$gt:40}}}]);

{ "TotalSalary" : 78, "EmpName" : "suku" }

d)$limit:- It limits the maximum no.of documents passed to next stage in pipeline.

Syntax:{$limit:<positive number>

Example:-

------------

> db.mca.find();

{ "\_id" : ObjectId("6185e323895a7fdfd4de05a7"), "name" : "suku", "age" : 37 }

{ "\_id" : ObjectId("6185e32a895a7fdfd4de05a8"), "name" : "suku2", "age" : 39 }

{ "\_id" : ObjectId("6185e338895a7fdfd4de05a9"), "name" : "suku3", "age" : 40 }

{ "\_id" : ObjectId("6185e340895a7fdfd4de05aa"), "name" : "suku", "age" : 41 }

> db.mca.aggregate([{$group:{\_id:"$name",TotalSalary:{$sum:"$age"}}}, {$sort:{TotalSalary:-1}},{$limit:1}]);

{ "\_id" : "suku", "TotalSalary" : 78 }

e)$skip:-It skips the no.of documents and pass the rest of documents to next stage in pipeline.

EX:- {$skip:positivenumber}

Example:

---------

> db.mca.find();

{ "\_id" : ObjectId("6185e323895a7fdfd4de05a7"), "name" : "suku", "age" : 37 }

{ "\_id" : ObjectId("6185e32a895a7fdfd4de05a8"), "name" : "suku2", "age" : 39 }

{ "\_id" : ObjectId("6185e338895a7fdfd4de05a9"), "name" : "suku3", "age" : 40 }

{ "\_id" : ObjectId("6185e340895a7fdfd4de05aa"), "name" : "suku", "age" : 41 }

> db.mca.aggregate([{$group:{\_id:"$name",TotalSalary:{$sum:"$age"}}}, {$sort:{TotalSalary:-1}},{$limit:1}]);

{ "\_id" : "suku", "TotalSalary" : 78 }

> db.mca.aggregate([{$group:{\_id:"$name",TotalSalary:{$sum:"$age"}}}, {$sort:{TotalSalary:-1}},{$skip:1}]);

{ "\_id" : "suku3", "TotalSalary" : 40 }

{ "\_id" : "suku2", "TotalSalary" : 39 }

f)$out:- It takes the document returned by pipeline and writes them to collection.

{$out:{db:”db-name”,coll:”coll-name”}}

> db.mca.find();

{ "\_id" : ObjectId("6185e323895a7fdfd4de05a7"), "name" : "suku", "age" : 37 }

{ "\_id" : ObjectId("6185e32a895a7fdfd4de05a8"), "name" : "suku2", "age" : 39 }

{ "\_id" : ObjectId("6185e338895a7fdfd4de05a9"), "name" : "suku3", "age" : 40 }

{ "\_id" : ObjectId("6185e340895a7fdfd4de05aa"), "name" : "suku", "age" : 41 }

> show dbs

Library 0.000GB

admin 0.000GB

config 0.000GB

local 0.000GB

sample 0.000GB

> db.mca.aggregate([{$group:{\_id:"$name",TotalSalary:{$sum:"$age"}}},{$out:{db:"sample",coll:"suku"}}]);

> use sample

switched to db sample

> db.suku.find()

{ "\_id" : "suku3", "TotalSalary" : 40 }

{ "\_id" : "suku", "TotalSalary" : 78 }

{ "\_id" : "suku2", "TotalSalary" : 39 }

>