MONGO DB

MONGO IS CASE SENSITIVE DATABASE QUERY LANGUAGE.

1.CREATE A DATABASE BY USING “USE”

USE \_ \_ \_DATABASE NAME\_ \_ \_ \_ \_

:- THIS IS SOMETHING WHEN THE DATABASE YOU’VE GIVEN IS EXIST THEN CONTROL WILL COME IN THAT PARTICULAR DATABASE, IF NOT AVAILABLE THEN, IT WILL CREATE A NEW DATABASE.

2. COLLECTION . THESE ARE LIKE TABLES IN MONGO DB.

AND THIS WILL CONTAIN THE KEY, VALUE PAIR.

KEY IS THE COLUMN NAME AND PAIR IS ITS VALUE. FOR EXAMPLE(‘EID: 1001). EID = KEY, 1001-= PAIR.

3.- LETS CREATE A DATABASE

####################################################################################################################

use lb23

\* - db.stu.insert({“rno”: 1, “name”: “kaushal kashyap”, “age”:12})

\*- db.stu.insertOne({“rno”: 1, “name”: “kaushal kashyap”, “age”:12})

= insertOne will return the object id of newly inserted data.

\*- db.stu.insertMany([{“rno” : 7 , “name” : “monica”}, {“rno” : 2 , “name”: “sonica”, “class” : “7th”}])

= inserting more than one documents(data) at a same time.

= this save work exact same as insert . It also use for inserting data.

\* db.stu.save({‘rno’:5,’name’:’Kaushal Kashyap’, ‘city’:’patna’})

\*db.stu.save({‘\_id’:ObjectId(“63233fa3ce”)})

Viola! Data inserted……

###################################################################################################################

commands:

1,- show dbs -it is basically for showing all the database inside that repository.

2.- db.stu.find() -it is to retrieve and show the data.

db.stu.find().pretty(); it also retrieve and show widely the documents(data). Or can say (“formatted Output”).

3.- db.dropdatabase() - will delete the whole database. But before deleting first check in which database you are in by using keyword ‘ db’.

4.- db.createCollection(“stu”) - to create a collection “stu”.

5.- show collections - it will show all the collection inside that database

6.- db.createCollection(“stu2”,{“capped”: true, size: 5000, max : 5}) = it follows last in first out crieteria.

7- db.stu.drop() = dropping the collection “stu”.

#####################################################################################################################

Update documents

# FOR A SINGLE DATA UPDATION USE UPDATEONE.

\* db.stu.updateOne({'EID': 1025},{$set:{'ADDRESS' : 'B302 PRAGYA APARTMENTS, DWARKA,DELHI','PHONE':9987456521}});

# FOR BELOW, IT IS SAYING THAT CHECK WHRER 'DEPT' IS 'HR' THEN SET 'DESI' TO SR. ASSOCIATE.

# IT WILL UPDATE ALL WHERE DEPT IS HR.

\* db.salary.updateMany({$and: [{"DEPT" :"HR"},{"DESI" :"ASSOCIATE"}]},

{$set:{"DESI" : "SR. ASSOCIATE"}});

\* db.salary.updateMany({$and: [{"DEPT" :"HR"},{"DESI" : "ASSOCIATE"}]},{$set:{"DESI" : "SR. ASSOCIATE"}});

# BELOW WITHOUT GIVING ANY CONDITION. UPDATE THE WHOLE COLUMN "CLASS" TO 8TH.

\* db.stu2.updateMany({},{"$set" : {"class" : "8th"}});

\* db.stu.replaceOne({"rno" : 2},{"rno" : 3 ,"name" : "ajay kumar","age" : 16});

\*

#####################################################################################################################

DELETE DOCUMENTS

\* db.stu.deleteOne({'rno': 5}); -- it will delete only one document specified under the condition.

\* db.emp.deleteOne({“name” : “ravi”});

\* db.collection.deleteMany({'dept':'HR'}) -- it will delete all the document where dept is hr.

\* db.stu.deleteMany({}) -- it will delete all the documents inside that collection.

\* db.emp2.deleteMany({ “SALARY”:{$lt:91280}})

\* db.sal.remove({“dept”: “MIS”}) --- it work similar as deleteMany().

\* db.sal.remove({“dept”: “ HR”},1) – it work similar as deleteOne().

\* db.sal.remove({“dept”: “ HR”},true) -- it work similar as deleteOne()..

###################################################################################

Operator in mongodb

8- Conditional statement . Like Inside a emp collection I want to see all “Hr” department employee details.

For that:

db.sal.find({“dept” : “HR”})

9- MORE THAN ONE CONDITION . THEN USE $ AND OPERATOR.

db.sal.find({ $and: [{ “dept” : “ops” }, { “desi” : “manager”}]})

db.sal.find({$and: [ {“dept”: “temp”},{“desi”: “associate”}]})

through this we can give as many condition as we want.

10- USING “$ OR” OPERATOR FOR CONDITIONAL QUESTION.

db.sal.find({$or : [{“dept” : “MIS”},{“desig” : “Associate”}]})

$or means either of any one condition or all condition is true.

\* $and operator

db.sal.find({$and: [{‘dept’ : ‘ops’},{‘desi’:’Manager’}]})

db.sal.find({‘dept’ : ‘hr’, ‘desi’ : ‘manager’})

db.sal.find({‘dept’ : ‘ops’, ‘desi’ : ‘manager’})

( $or operator;

db.sal.find({$or: [{‘dept’ : ‘hr’},{‘dept’: ‘IT’}]});

db.sal.find({$or: [{‘dept’ : ‘hr’},{‘dept’: ‘mis’}]});

db.sal.find({$or: [{ “dept” : “ops”},{“dept” : “hr”}]});

db.sal.find({$and: [{‘desi’: ‘associate’}.{$or: [{‘dept’ : “ops”},{‘dept’: “hr”}]}]})

\* Not Equals to :-

db.sal.find({“dept”: {$not : {$eq : “temp”}}}) – mean other which dept!= temp

\* db.sal.find({$nor: [{“dept”: “temp”},{“desi” : “associate”}]})

means neither department should be temp nor the desi should be associate.

###################################################################################

Comparison operator

\* $eq = equals to.

db.stu.find({“dept” : {$eq : “mis”}});

\* db.stu.find({$and : [{“dept”: {$eq : “OPS”}},{“DESI”: {$eq : ‘manager’}}]})

\* db.emp.find({$or: [{“key” : “value”},{“key” : “value”})

\* db.emp.find({“DEPT”: {$NOT: {$eq: “hr”}}});

\* db.sal.find({$nor: [{“desi”: “associate”}, {“dept”: “hr”}]});

\* eq( equals to ):- $eq: Equality Operator same as where dept = ‘hr’ in SQL

\* db.sal.find({"DEPT" :{$eq : "HR"}});

\* db.sal.find({"DEPT" : "HR"});

\* - $ne: Not equal to Operator same as where dept <> ‘hr’ in SQL

-- db.sal.find({"DEPT" :{$ne : "HR"}});

\* $lt: less than Operator same as where salary< 100000 in SQL

Syntax: {<key>:{$lt:<value>}}

Eg: > db.sal.find({"SALARY" :{$lt : 50000}});

\* $lte: less than or equal to Operator same as where salary<= 50000 in SQL

Syntax: {<key>:{$lte:<value>}}

Eg: > db.sal.find({"SALARY" :{$lte : 50000}});

\* $gt: greater than or equal to Operator same as where salary>= 50000 in SQL

Syntax: {<key>:{$gt:<value>}}

Eg: > db.sal.find({"SALARY" :{$gt : 50000}});

\* $gte: greater than or equal to Operator same as where salary >= 50000 in SQL

Syntax: {<key>:{$gte:<value>}}

Eg: > db.sal.find({"SALARY" :{$gte : 50000}});

\* $in: represents values in an array same as where dept in (‘hr’, ‘it’, ’admin’)

Syntax: {<key>:{$in:[<value1>, <value2>,......<valueN>]}}

Eg: > db.sal.find({"DEPT" :

{$in :

["HR", "IT", "TEMP"]

}

});

\* $nin: represents values not in an array same as where NOT dept in (‘hr’, ‘it’,

’admin’)

Syntax: {<key>:{$nin:[<value1>, <value2>,......<valueN>]}}

Eg: > db.sal.find({"DEPT" :{$nin : ["HR", "IT", "TEMP“,”ADMIN”]}});

Operators Examples:

\* db.orders.find({"Category" : {"$eq" : "Technology"},"Sub-Category" : {"$eq" : "Phones"}}).count()

> db.salary.find({"$and" : [{"DEPT" : {"$eq" : "HR"}}, {"DESI" : {"$eq" : "ASSOCIATE"}}]})

> db.salary.find({"$and" : [{"DEPT" : "HR"}, {"DESI" : "ASSOCIATE"}]})

> db.salary.find({"$and" : [{"$or" : [{"DEPT" : "HR"}, {"DEPT" : "MIS"}]},{"$or" : [{"DESI" :

"MANAGER"}, {"DESI" : "SR. ASSOCIATE"}]}]})

> db.salary.find({"SALARY" :{"$not" : {"$gt" : 100000}}})

-- Importing Data

d: Specifies what database to use. We used the demo database.

-c: Specifies what collection to use. We used a sal collection.

--type: Specifies the type of file to import. json, csv, or tsv. We are using csv

--headerline: Specifies that the first row in our csv file should be the field names.

--drop: Specifies that we want to drop the collection before importing documents to avoid

duplicate documents.

To import a csv [file:-](../../../../../-)

>

**>mongoimport -d demo2 -c sal --type csv --file C:\Users\Raj\Desktop\Salary.csv --**

**headerline --drop**

# To get the count of the records:

\* db.stu.find().count()

**Projection**

**projection are for filtering data to display or should act on it..**

For better understanding looks this way:

db.collection.field({condition},{projection})

\* db.sal.find({},{“EID” : 1, “dept”: 1, “salary”: 1})

\* db.sal.find({},{“desi”: 0 })

suppose I want to see only their eid, salary, and should belongs to hr desi..

\* db.sal.find({“dept” : “hr”), {“EID”: 1, “salary”: 1})

**Limiting the documents**

\* db.sal.find().limit(10);

**Skipping the documents;**

\* db.sal.find().limit().skip()

**Sorting the documents**

**\*** db.sal.find().sort({“salary”: 1})

- sorting on the basis of desi hr in salary.

\* db.sal.find({“desi”: “hr”}).sort({“salary”: 1}) -- in ascending order

\* db.sal.find({“desi”: “hr”}).sort({“salary”: -1}) -- in descending order

\* db.sal.find({“dept”: “hr”},{“desi”: 0}).sort({“salary”: 1}).limit(3) -- sort on the basis of dept=hr.

###################################################################################

**Indexing**

\* db.emp.createIndex({“city”: 1})

to see the index -

db.emp.getIndexes()- To DISPLAY THE INDEXES.

\* compound index

db.emp.createIndex({“city”: 1, “addr”: 1}); -- compound is basically for indexing more than 1 column.

\* time to leave (ttl) –db.emp.createIndex({“EId ”: 1},{expiredafterseconds: 600})-- after that 600s index will automatically be destroyed.

\* unique indexing

db.sal.createIndex({“EID”: 1, }, {unique: true}) – using unique keyword eid will always have unique value. There will be no duplicacy.

Note: Through indexing you can stop duplicacy in database.

Dropping the indexes:

\* db.emp.dropIndexes({“emp”: -1}) – condition dropping

\* db.emp.dropIndexes({}) -- to drop all the indexes.

Note: a collection can have maximum 64 indexes.

###################################################################################

**Aggregation(“ brought together ”)**

Aggregation is basically for perform operation stage by stage. One’s calculation’s value is used as another stage source to perform operation.

db.sal.aggregate([{$group: { \_id : “dept”, “TC”: {$sum : “$salary”}}}]);

here we are gouping the department on the basis of sum of salary.

\* db.sal.aggregate([{$ group: {\_id : “$dept”, “TC”: {$sum: “$salary”},”tm”: {$sum: 1}}}]) -- grouping as per department with the sum of every department salary and team members,

\* db.sal.aggregate([{$ group: {\_id : “$dept”, “TC”: {$sum: “$salary”},”tm”: {$sum: 1},”avgsal”: {“avg”: “$salary”} }}]) -- grouping as per dept with sum of each department salary and their average salary.

\* db.sal.aggregate([{$ group: {\_id : “$dept”, “TC”: {$sum: “$salary”},”tm”: {$sum: 1},”avgsal”: {“avg”: “$salary”} ,”maxsal”:{$max:”$salary”},”Minsal”: {“$min”: “$salary”}}}]) -- grouping as per department with each department salary with its sum, Average salary, Minimum salary, Maximum Salary.

$sum - Sums up the defined value from all documents in the

collection

$avg - Calculates the average of all given values from all documents

in the collection

$min - Gets the minimum of the corresponding values from all

documents in the collection

$max - Gets the maximum of the corresponding values from all

documents in the collection

Atomic Operations

# findAndModify() – search for the document and modify it.

# query: {} – specify the search criteria

# update: {} – specify the updation in the document

# $inc: - Increments the value of the field by the specified amount.,

# $push: - Adds an item to an array.

# first create an order which contains product details, stock details and inside an array it should contain Buyer details.

db.order.insert({“\_id”: 1,”PID”: “C001”,”PDESC”:”Dell Mouse”, “Price”: 200, “TotalStk”: 10, “BalStk”: 8, “SQTY”:10, “Purchased By”: [{“CID”: “C001”, “Name”: “Amit”, “Pdate”: “01-DEC-2022”},{“CID”: “C002”,”Name”: “Ravi”, “Pdate”: “02-Sept-2022”}]})

# to update the stock after purchasing the product, we use “findAndModify” to autoupdate after purchasing.

db.order.findAndModify({query: {“\_id”: 1, “Balstk”: {$gt: 0}}, update: {$inc: {“Balstk”: -1}$push: {“Purchased By”: {“CID”: “C001”, “Name”: “Amit”, “Pdate”: “03-sept-2022”}} }})

Output:

"\_id" : 1,

"PID" : "P001",

"PDESC" : "Dell Mouse",

"Price" : 200,

"TotalStk" : 10,

"Balstk" : 8,

"SQTY" : 10,

"Purchased BY" : [

{

"CID" : "C001",

"Name" : "Amit",

"PDate" : "01-DEC-2022"

},

{

"CID" : "C002",

"Name" : "Ravi",

"PDate" : "02-Sept-2022"

}

],

"Purchased By" : [

{

"CID" : "C001",

"Name" : "Amit",

"PDate" : "03-sept-2022"

}

]

Again:

db.order.findAndModify({query: {“\_id”: 1, “Balstk”: {$gt: 0}}, update: {$inc: {“Balstk”: -1}$push: {“Purchased By”: {“CID”: “C001”, “Name”: “Amit”, “Pdate”: “03-sept-2022”}} }})

Output:

"\_id" : 1,

"PID" : "P001",

"PDESC" : "Dell Mouse",

"Price" : 200,

"TotalStk" : 10,

"Balstk" : 7,

"SQTY" : 10,

"Purchased BY" : [

{

"CID" : "C001",

"Name" : "Amit",

"PDate" : "01-DEC-2022"

},

{

"CID" : "C002",

"Name" : "Ravi",

"PDate" : "02-Sept-2022"

}

],

"Purchased By" : [

{

"CID" : "C001",

"Name" : "Amit",

"PDate" : "03-sept-2022"

}

]

As, you can see the stock is getting updated on every purchase.

Or you can also find the update in using find().

db.order.find().pretty()

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

**Data Modeling**

Databases are made to store data. RDBMS comes with features which is reduce duplicacy, security, no confliction, and Data should be structured.

But for a single entity data, the information stored at many places . When you go to retrieve the data info. You will be needing to access in many databases, tables.etc. THIS IS CALLED NORMALISED DATA MODEL.

Example: consider the inventory query where, customer , order, stock, product were all was different table . If you want to retrieve the data which may need from more than one table. You will have to access on those required tables. . this is called Normalised data model.

EMBEDDED MODEL:- It is kind of model where all required data will be stored in one place in embedded manner. So the retrieving will be easy … It Is also called DeNormalised model

Example:- db.emp2.insertOne({eid: "e0001",PD: {fn : "Amit", ln : "kumar", dob : "10-may-1990"},

contact: {ph : "98899787690", email :"akumar@gmail.com"},addr: {area : "sector 5 Dwarka", city : "delhi"},off: {dept : "ops", desi : "manager" , salary : 90000}});

Referencing:- Inserting the object Id of one document in another document is known as referencing.

Basically, In Referencing we give the value to key as another’s ObjectId whose you want to call.

So, when you call it , it will display this field information as well as ObjectId referred data.

“aid”: ObjectId(“60mngjh4545bnfhg546”);

Lookup Aggregation:- it adds an array of related data from the other document. It perform an equality match between a field from the input documents with a field from the documents of the "joined" collection.

Examples: db.emp3.aggregate([{$lookup:{from: "salary" ,localField: "EID",foreignField: "EID",as: "SalDetails" }}]);

How it works.:-

({$lookup: {from:”collection name from where you want to fetch the data”,localFeild:” name of the key(Column)”, foriegnField: “\_id” as “By”}}) -- here foreignField will contain \_id (holds key id which you want to call ).

Relationship in MongoDB

Here in mongo Relationship is understable here as reference.

One – to – One(1:1): A document contain a single reference id of another.

One – to --Many (1:M): A document contain a Many reference id of another.

Many- to Many (M:M): Many documents contain many reference

Many- to – One(M:1): Many documents contain Single reference.

Data Types in MongoDB

In MongoDB data is data representation is done in JSON (JavaScript Object Notation)

document format which is binary encoded and is termed as BSON. MongoDB supports

many data types.

Integer − This type is used to store a numerical value.

> db.testdt.insert({"integer" :125});

Boolean − This type is used to store a boolean (true/ false) value.

> db.testdt.insert({"registered" :true});

Double − This type is used to store floating point values.

> db.testdt.insert({"amount" : 3745.95});

String − This is the most commonly used datatype to store the data.

> db.testdt.insert({“greeting" : “Welcome to MongoDB”});

Arrays − This type is used to store arrays or list or multiple values into one key.

> var courses = ["SQL" ,"PBI", "MongoDB"]

> db.testdt.insert({"module" : courses});

Object − This datatype is used for embedded documents.

> var hrs = {"SQL" : 25, "PowerBi" :20 , "MongoDB" : 15};

> db.testdt.insert({"Duration" : hrs});

Date − This datatype is used to store the date.

> var d1 = Date();

> var d2 = ISODate();

> var JD = ISODate("2021-05-01");

> db.testdt.insert({"Stringdate" : d1, "ISODate" : d2 , "JoiningDate": JD});

**MongoDB GridFS**

GridFS is a frame work to store & access large set of data. It divides

the data into chunks and store then into different documents.

-API Provided by MongoDb for storing large files such as audio, video

and images.

-Package that can be plucked into any application to make storing

large files easier

Provides a way for storing large files in database instead of in the file

system.

Problem: In MongoDB document size is limited to  
16 MB.

Gridfs Solves the size limitation problem

1. Breaks the files to smaller managable chunks

2. Stores these chunks of data in one collection called fs.chunks

3. Stores the information about the whole file itself in another collection

called fs.files

4. Connects these documents by properties that are references to each

other

**MongoDB GridFS**

fs.chunks collection

1. The size of each chunk is 255KB

2. No. of chunks created depends on

the file size

3. Chunks stores the actual data.

4. Each chunk is linked to the fille

information by “files\_id” property.

5. The “files\_id” points to a

document that is stored in fs.files

collection

**MongoDB GridFS**

fs.files collection contains the information about the file

1. File name

2. Average size of each chunk

3. Upload date

4. Size of file (in bytes)

5. File metadata