UNIT:5 USING MONGO DB

MongoDB Introduction:

MongoDB is a cross-platform, document oriented database that provides, high performance, high availability, and easy scalability.

Purpose of Building MongoDB:

The primary purpose of building MongoDB is:

1. Scalability

2. Performance

3. High Availability

4. Scaling from single server deployments to large, complex multi-site architectures.

5. Develop Faster

6. Deploy Easier

7. Scale Bigger

Where do we use Mongo DB?

MongoDB is preferred over RDBMS in the following scenarios:

1. Big Data: If we have huge amount of data to be stored in tables, think of MongoDB before RDBMS databases. MongoDB has built-in solution for partitioning and sharding database.

2. Unstable Schema: Adding a new column in RDBMS is hard whereas MongoDB is schema-less. Adding a new field does not effect old documents and will be very easy.

3. Distributed data Since multiple copies of data are stored across different servers, recovery of data is instant and safe even if there is a hardware failure.

MongoDB Basics:

MongoDB works on concept of collection and document.

Documents

 MongoDB is a document-oriented database.

 A document is a data structure composed of field and value pairs.

 The values of fields may include objects, arrays, and arrays of objects and so on.

 MongoDB documents are similar to JSON objects, so it is easy to think of them as JavaScript objects.

 MongoDB document has support not only for the primitive data types—Boolean, numbers, and strings—but also other common data types such as dates, timestamps, regular expressions, and binary data.

Example:

An Invoice object may look like this:

{

"invoiceNumber" : 1234,

"invoiceDate" : ISODate("2022-19-12"),

"billingAddress" : {

"name" : "Ram electronics.", "line1" : "106 High Street", "city" : "kavali", "zip" : "524201"

},

"items" : [

{

"description" : "Compact Flourescent Lamp", "quantity" : 4,

"price" : 12.48

},

{

"description" : "Whiteboard", "quantity" : 1,

"price" : 5.44

}

]

}

In this document, there are numbers, strings, and a date data type. Further, there is a nested object (billingAddress) and an array of objects (items).

Collections

 A collection is like a table in a relational database. it is a set of documents.

 The collection can have a primary key and indexes.

 A primary key is mandated in MongoDB, and it has the reserved field name \_id. Even if \_id field is not supplied when creating a document, MongoDB creates this field and auto-generates a unique key for every document.

 The auto-generated ID can be used as is, since it is convenient and guaranteed to produce unique keys even when multiple clients are writing to the database simultaneously.

 MongoDB uses a special data type called the ObjectId for the primary key.

 The \_id field is automatically indexed.

 Indexes are used to efficiently access a subset of documents in a collection.

 MongoDB does not require to define a schema for a collection. The only requirement is that all documents in a collection must have a unique \_id.

Databases

A database is a logical grouping of many collections. . Most database operations read or write from a single collection. Most database operations read or write from a single collection.

A database connection is restricted to accessing only one database, so to access multiple databases, multiple connections are required.

Query Language

MongoDB query language is made up of methods to achieve various operations. The main methods for read and write operations are the CRUD methods. Other methods include aggregation, text search, and geospatial queries.

All methods operate on a collection and take parameters as JavaScript objects that specify the details of the operation. Each method has its own specification.

For example, to match all documents with the field invoiceNumber that are greater than 1,000, the following query filter can be used:

{ "invoiceNumber": { $gt: 1000 } }

MongoDB encourages denormalization, that is, storing related parts of a document as embedded subdocuments rather than as separate collections (tables) in a relational database.

Installation

There are many services available to access to MongoDB. Some of them are:

• MongoDB Atlas (https://www.mongodb.com/cloud/atlas): I refer to this as Atlas for short. A small database (shared RAM, 512 MB storage) is available for free.

• mLab (previously MongoLab) (https://mlab.com/): mLab has announced an acquisition by MongoDB Inc. and may eventually be merged into Atlas itself. A sandbox environment is available for free, limited to 500 MB storage.

• Compose (https://www.compose.com): Among many other services, Compose offers MongoDB as a service. A 30-day trial period is available, but a permanently free sandbox kind of option is not available.

If we choose to install MongoDB on local computer look up the installation instructions, which are different for each operating system.

Install MongoDB by following the instructions at the MongoDB website

(https://docs.mongodb.com/manual/installation/ or search for “mongodb installation” in your search engine).

Choose MongoDB version 3.6 or higher, preferably the latest, as some of the examples use features introduced only in version 3.6. Most local installation options let you install the server, the shell, and tools all in one. Check that this is the case; if not, you may have to install them separately.

After a local installation, ensure that you have started MongoDB server (the name of the daemon or service is mongodb), if it is not already started by the installation process. Test the installation by running the mongo shell like this:

$ mongo

On a Windows system, you may need to append .exe to the command. The command may require a path depending on your installation method. If the shell starts successfully, it will also connect to the local MongoDB server instance. You should see the version of MongoDB printed on the console, the database it is connecting to (the default is test), and a command prompt, like this, if you had installed MongoDB version

4.0.2 locally:

MongoDB shell version v4.0.2

connecting to: mongodb://127.0.0.1:27017 MongoDB server version: 4.0.2

>

The message you see can be slightly different from this, especially if you have installed a different version of MongoDB. But you do need to see the prompt > where you can type further commands. If, instead, you see an error message, revisit the installation and the server starting procedure.

The Mongo Shell

The mongo shell is an interactive JavaScript shell, very much like the Node.js shell. In the interactive shell, a few non-JavaScript conveniences are available over and above the full power of JavaScript.

To work with MongoDB, we need to connect to a database. The command to show the current databases is:

> show databases

a fresh local installation of MongoDB, this is what we will see:

admin 0.000GB

config 0.000GB

local 0.000GB

These are system databases that MongoDB uses for its internal book keeping, etc. We will not be using any of these to create our collections.

To identify the current database, the command is:

> db

The default database a mongo shell connects to is called test .

The command to know collections available in the database.

> show collections

switch to a database called issuetracker instead of using the default database:

> use issuetracker

output

switched to db issuetracker

 insertOne() method is used to add new document in the collection .

Example:

insert a new document in the employee’s collection using the insertOne() method is:

db.employees.insertOne({ name: { first: ‘vijay', last: ‘Bhaskar' }, age: 35 })

 Some important methods are:

1. insertOne()

2. insertMany()

3. insert()

4. count()

5. createIndex()

6. addIdIfNeeded()

7. getWriteConcern()

8. aggregate()

9. group()

10. bulkWrite()

11. groupcmd()

12. constructor()

13. hasOwnProperty()

14. convertToCapped()

15. hashAllDocs()

16. convertToSingleObject( )

17. help()

18. copyTo()

19. initializeOrderedBulkOp()

20. initializeUnorderedBulkOp()

21. createIndexes()

22. dataSize()

 find() method is used to check whether the document has been created in the collection or not.

Example:

db.employees.find()

 pretty() method is used to display output in formatted way.

Example:

db.employees.find().pretty()

Output:

{

"\_id" : ObjectId("5bbc487a69d13abf04edf857"), "name" : {

"first" : "John",

"last" : "Doe"

},

"age" : 44

}

**MongoDB CRUD Operations:**

The CRUD operations are:

* C-Create
* R-Read
* U-Update
* D-Delete

**Create:**

Create (or insert) operations add new documents to a collection. There are two ways to add new documents to a collection:

• insertOne()

• insertMany()

insertOne()

The insertOne() command allows you to insert one data object at a time into the collection. It takes some options in key/value pairs separated by a comma thus establishing the schema.

The command db.collection.insert() will perform the insert operation into the collection of the document.

Example:

>db.studentdetails.insert({

"enrollno": "501",

"name": "M Vijaya Bhaskar",

"college": "VITS Engineering College",

})

It will insert a document in MongoDB

insertMany()

insertMany() method is used to add multiple documents at a time.

**Syntax**:

db.collection.insertMany(

[ <document 1> , <document 2>, ... ],

{

writeConcern: <document>,

ordered: <boolean>

}

)

It returns the following document containing:

1. A boolean acknowledged as valid if the operation ran with write concern or false if write concern was disabled.

2. An array of \_id for each successfully inserted documents.

Example:

>db.studentdetails.insertMany([{

"enrollno": "502",

"name": "Mahesh",

"college": "VITS Engineering College"

},{

"enrollno": "503",

"name": "Siva",

"college": "VITS Engineering College"

}])

**Note:**

MongoDB automatically creates the primary key, which was a special data type called ObjectID.

We can create our own ID also.

Example:

> db.employees.insertOne({

\_id: 1,

name: { first: 'John', last: 'Doe' }, age: 44

})

output:

{ "acknowledged" : true, "insertedId" : 1 }

**Read:**

Read operations retrieve documents from a collection. There are 2 methods to retrieve documents from a collection. Those are:

• find()

• findOne()

find():

find() method is used to get all the documents from a collection.

**Syntax**:

db.collection.find()

Example:

Command to display all the documents of Studentdetails is as follows:

>db. studentdetails

findOne()

findOne() method is used to retrieve information from a collection by specify any filter or a criteria.

**Syntax**:

db.collection.find(query)

Example:

Command to display the documents of Studentdetails containing enrollno as "502" is as follows:

>db. Studentdetails.find({"enrollno": "502"})

**Update**

Update operations modify existing documents in a collection. There are three ways to update documents of a collection:

• db.collection.updateOne()

– Updates one field in the document where the given criteria or filter meets the condition. Updating a field will not remove the old field instead a new field will be added to the document.

• db.collection.updateMany()

– Updates all fields in the document where the given criteria or filter meets the condition.

• db.collection.replaceOne()

– Replace the entire document. It will replace the old fields and values with new ones.

Example

For example, if we have the following document:

{

"\_id" : ObjectId("1"),

"model" : 2005

}

Using:

replaceOne({"\_id" : ObjectId("1")}, { "new\_model" : 2020})

will result in:

{

"\_id" : ObjectId("1"),

"new\_model" : 2020

}

While using:

updateOne({"\_id" : ObjectId("1")}, {＄set: { "new\_model" : 2020}})

will result in:

{

"\_id" : ObjectId("1"),

"model" : 2005,

"new\_model" : 2020

}

While using:

updateMany({"\_id" : ObjectId("1")}, {＄set: { "name" : "NewName"}, ＄set: { "new\_model" : 2020}})

will result in:

{

"\_id" : ObjectId("1"),

"model" : 2005,

"new\_model" : 2020

"name" : "newName"

}

**Delete:**

Delete operations delete documents from a collection. There are two methods to delete documents of a collection:

• db.collection.deleteOne()

• db.collection.deleteMany()

deleteOne()

This method removes only the first document matched by the query filter document.

Example

remove only one car having the model “2013” from the cars collection:

db.cars.deleteOne(

//deletes one car having model "2013"

{ "model": "2013" }

)

deleteMany()

• deletes multiple objects removes all the matched by the query filter document at once.

Example

To remove all cars having the model “2013” from the cars collection

db.cars.deleteMany(

//delete all cars having model "2013"

{ "model": "2013" }

)

Schema Initialization

The mongo shell is not only an interactive shell, but is also a scripting environment.

scripts can be used to write to perform various tasks such as schema initialization and migration.

Example For Schema Creation:

const NewSchema=new mongoose.Schema({

name:String,

rollno:Number

});

Program to connect, creating a Database and storing data into MongoDb

const express=require('express');

const app=express();

const mongoose=require('mongoose');

app.listen(5000, ()=>{console.log('connection listened on 50000')})

mongoose.set('strictQuery', true);

mongoose.connect("mongodb://localhost:27017",(err)=>{

if(!err) console.log('database connected')

else console.log('database error');})

const NewSchema=new mongoose.Schema({

name:String,

rollno:Number

});

const newModel=new mongoose.model("collections",NewSchema);

const data=new newModel({name:'bhaskar',rollno:501});

data.save();

console.log(data);

**MongoDB Node.js Driver**

Procedure to use MongoDB With Node.js Driver is as follows:

1. Install the MongoDB Node.js Driver

 Create a new file named mongodb-nodejs and go to the current folder mongodb-nodejs with CLI (Command Line Interface) as follows:

mkdir mongodb-nodejs

cd mongodb-nodejs

 Create a new node project with npm that add the package.json file inside mongodb-nodejs folder

npm init –y

 Install the mongodb driver for nodejs to use the MongoDB database with nodejs

npm install mongodb –save

 mongodb driver helps us to connect and easily manage queries in MongoDB with nodejs

2. Connecting to The Local MongoDB Database

 Define a path to store data for MongoDB on the local machine. We will add the path C:\Program Files\MongoDB\data\db. Also make sure that the specified path/folder exists.

mongod --dbpath 'C:\Program Files\MongoDB\data\db'

--dbpath to add a path for the MongoDB database and start the server locally.

3. Configuring The MongoDB Node.js Connection

 Create a file named server.js and add the following code to the server.js file

const { MongoClient } = require('mongodb')

// Create Instance of MongoClient for mongodb

const client = new MongoClient('mongodb://localhost:27017')

// Connect to database

client.connect()

.then(() => console.log('Connected Successfully'))

.catch(error => console.log('Failed to connect', error))

 Run the node server.js command and We will see the following output

$ node server.js

Connected Successfully!

4. Closing The Connection

 Change the code of server.js file as follows

const { MongoClient } = require('mongodb')

// Create Instance of MongoClient for mongodb

const client = new MongoClient('mongodb://localhost:27017')

// Connect to database

client.connect()

.then(() => {

console.log('Connected Successfully!')

//Close the database connection

console.log('Exiting..')

client.close()

})

.catch(error => console.log('Failed to connect!', error))

 close() method to disconnect the database from the node app.

 Node app will exit from the MongoDB instance after connecting the node to the database.

 Run node server.js

$ node server.js

Connected Successfully!

Exiting..

Example program to connect to school database is as follows:

Connect-mongo-client.js

const MongoClient = require('mongodb').MongoClient

// Connect URL

const url = 'mongodb://127.0.0.1:27017'

// Connect to MongoDB

MongoClient.connect(

url,

{

useNewUrlParser: true,

useUnifiedTopology: true

},

(err, client) => {

if (err) {

return console.log(err)

}

// Specify the database you want to access

const db = client.db('school')

console.log(`MongoDB Connected: ${url}`)

}

)

To execute the above code, type the following command:

$ node connect-mongo-client.js

MongoDB Connected: mongodb://127.0.0.1:27017

Reading Data from MongoDB:

To read data from mongoDb first we need to have a database and collection in MongoDb.

Example:

Database Name: college

Collection Name: staff details

It contains the following information:

Program to read data from college database is as follows:

Server.js

const express=require('express');

const app=express();

const mongoose=require('mongoose');

const {MongoClient}=require('mongodb')

app.listen(5000, ()=>{console.log('connection listened on 50000')})

mongoose.set('strictQuery', true);

const url="mongodb://localhost:27017";

const client=new MongoClient(url);

const database='college';

async function getData(){

let result =await client.connect();

db=result.db(database);

collection=db.collection('staffdetails');

let data=await collection.find({}).toArray();

console.log(data);

}

getData();

output:

run the server.js file as follows:

nodemon server.js

It displays output as follows:

connection listened on 50000

[

{

\_id: new ObjectId("63ae62115fc6e93dcf0a7801"),

name: 'Bhaskar',

Branch: 'CSE',

salary: 45000.674

},

{

\_id: new ObjectId("63ae62115fc6e93dcf0a7802"),

name: 'Vijay',

Branch: 'ECE',

salary: 29000.5

},

{

\_id: new ObjectId("63ae6c3b5fc6e93dcf0a7803"),

name: 'Rajesh',

Branch: 'ECE',

salary: 19000

}

]

Writing Data to MongoDB:

Consider a database and collection to write data into MongoDB.

Example:

Database Name: college

Collection Name: staff details

It contains the following information:

Program to write data into college database is as follows:

Server.js

const express=require('express');

const app=express();

const mongoose=require('mongoose');

var MongoClient = require('mongodb').MongoClient;

var url = "mongodb://localhost:27017/";

app.listen(5000, ()=>{console.log('connection listened on 5000')})

mongoose.set('strictQuery', true);

MongoClient.connect(url, function(err, db) {

if (err) throw err;

var dbo = db.db("college");

var myobj = { name: 'Prasad',Branch: 'Mech',salary: 22000.4 };

dbo.collection("staffdetails").insertOne(myobj, function(err, res) {

if (err) throw err;

console.log("1 document inserted");

console.log(dbo.collection("staffdetails").find({}))

db.close();

});

});

Output:

run the server.js file as follows:

nodemon server.js

It displays output as follows:

connection listened on 50000

1 document inserted

If we open MogoDB staffdetails collections one document is added at the end as follows:

**MongoDB - Aggregation**

Aggregation is the process of selecting data from a collection in MongoDB. It processes multiple documents and returns computed results.

Use aggregation to group values from multiple documents, or perform operations on the grouped data to return a single result.

Aggregation operations can be performed in two ways:

1. Using Aggregation Pipeline.

2. Using single purpose aggregation methods:

db.collection.estimatedDocumentCount(),

db.collection.count () and

db.collection.distinct().

**Aggregation Pipelines**

The aggregation pipeline is an array of one or more stages passed in the db.aggregate() or db.collection.aggregate() method.

Syntax:

db.collection.aggregate([ {stage1}, {stage2}, {stage3}...])

Aggregation framework processes the pipeline of stages on the collection data and gives output in the form we need.

Every stage receives the output of the previous stage, processes the data further, and sends it to the next stage as input data. Aggregation pipeline executes on the server can take advantage of indexes.

the possible stages in aggregation framework −

• $project − Used to select some specific fields from a collection.

• $match − This is a filtering operation and thus this can reduce the amount of documents that are given as input to the next stage.

• $group − This does the actual aggregation as discussed above.

• $sort − Sorts the documents.

• $skip − With this, it is possible to skip forward in the list of documents for a given number of documents.

• $limit − This limits the amount of documents to look at, by the given number starting from the current positions.

• $unwind − This is used to unwind document that are using arrays. When using an array, the data is kind of pre-joined and this operation will be undone with this to have individual documents again. Thus with this stage we will increase the amount of documents for the next stage.

Example:

employees collection:

Sample Data

db.employees.insertMany([

{

\_id:1,

firstName: "John",

lastName: "King",

gender:'male',

email: "john.king@abc.com",

salary: 5000,

department: {

"name":"HR"

}

},

{

\_id:2,

firstName: "Sachin",

lastName: "T",

gender:'male',

email: "sachin.t@abc.com",

salary: 8000,

department: {

"name":"Finance"

}

},

{

\_id:3,

firstName: "James",

lastName: "Bond",

gender:'male',

email: "jamesb@abc.com",

salary: 7500,

department: {

"name":"Marketing"

}

},

{

\_id:4,

firstName: "Rosy",

lastName: "Brown",

gender:'female',

email: "rosyb@abc.com",

salary: 5000,

department: {

"name":"HR"

}

},

{

\_id:5,

firstName: "Kapil",

lastName: "D",

gender:'male',

email: "kapil.d@abc.com",

salary: 4500,

department: {

"name":"Finance"

}

},

{

\_id:6,

firstName: "Amitabh",

lastName: "B",

gender:'male',

email: "amitabh.b@abc.com",

salary: 7000,

department: {

"name":"Marketing"

}

}

])

$match Stage

The $match stage is usually the first stage to select only the matching documents from a collection. It is equivalent to the find() method.

Example:

db.employees.aggregate([ {$match:{ gender: 'female'}} ])

It will return all documents where gender:'female' filed.

Output

[

{

\_id: 4,

firstName: 'Rosy',

lastName: 'Brown',

gender: 'female',

email: 'rosyb@abc.com',

salary: 5000,

department: { name: 'HR' }

}

]

$group Stage

the $group stage is used to group the input documents by the specified \_id expression and returns a single document containing the accumulated values for each distinct group.

Example:

db.employees.aggregate([

{ $group:{ \_id:'$department.name'} }

])

Output:

[ { \_id: 'Marketing' }, { \_id: 'HR' }, { \_id: 'Finance' } ]

The following calculates the number of employees in each department.

db.employees.aggregate([

{ $group:{ \_id:'$department.name', totalEmployees: { $sum:1 } }

}])

Output:

[

{ \_id: 'Marketing', totalEmployees: 2 },

{ \_id: 'HR', totalEmployees: 2 },

{ \_id: 'Finance', totalEmployees: 2 }

]

The following aggregation pipeline contains two stages.

db.employees.aggregate([

{ $match:{ gender:'male'}},

{ $group:{ \_id:'$department.name', totalEmployees: { $sum:1 } }

}])

Output:

[

{ \_id: 'Marketing', totalEmployees: 2 },

{ \_id: 'HR', totalEmployees: 1 },

{ \_id: 'Finance', totalEmployees: 2 }

]

$sort Stage

The $sort stage is used to sort the documents based on the specified field in ascending or descending order. The following sorts all male employees.

db.employees.aggregate([

{ $match:{ gender:'male'}},

{ $sort:{ firstName:1}}

])

Output:

[

{

\_id: 6,

firstName: 'Amitabh',

lastName: 'B',

gender: 'male',

email: 'amitabh.b@abc.com',

salary: 7000,

department: { name: 'Marketing' }

},

{

\_id: 3,

firstName: 'James',

lastName: 'Bond',

gender: 'male',

email: 'jamesb@abc.com',

salary: 7500,

department: { name: 'Marketing' }

},

{

\_id: 1,

firstName: 'John',

lastName: 'King',

gender: 'male',

email: 'john.king@abc.com',

salary: 5000,

department: { name: 'HR' }

},

{

\_id: 5,

firstName: 'Kapil',

lastName: 'D',

gender: 'male',

email: 'kapil.d@abc.com',

salary: 4500,

department: { name: 'Finance' }

},

{

\_id: 2,

firstName: 'Sachin',

lastName: 'T',

gender: 'male',

email: 'sachin.t@abc.com',

salary: 8000,

department: { name: 'Finance' }

}

]

The following pipeline contains three stages to sort the groupped documents.

db.employees.aggregate([

{ $match:{ gender:'male'}},

{ $group:{ \_id:{ deptName:'$department.name'}, totalEmployees: { $sum:1} } },

{ $sort:{ deptName:1}}

])

Output:

[

{ \_id: { deptName: 'Finance' }, totalEmployees: 2 },

{ \_id: { deptName: 'HR' }, totalEmployees: 1 },

{ \_id: { deptName: 'Marketing' }, totalEmployees: 2 }

]