PRACTICAL ASSIGNMENT - 1

PROBLEM STATEMENT -

Create a database with suitable example using MongoDB and implement
-Inserting and saving document (batch insert, insert validation)
- Removing document
-Updating document (document replacement, using modifiers, up inserts, updating multipledocuments, returning updated documents)
-Execute at least 10 queries on any suitable MongoDB database that demonstrates following:
Find and find One (specific values)
Query criteria (Query conditionals, OR queries, $not, Conditional semantics)
Type-specific queries (Null, Regular expression, Querying arrays)
$ where queries
Cursors(Limit,skip,sort, advanced query options)

THEORY -

MongoDB, the most popular NoSQL database, is an open-source document-oriented database. The term 'NoSQL' means 'non-relational'. It means that MongoDB isn't based on the table-like relational database structure but provides an altogether different mechanism for storage and retrieval of data. This format of storage is called BSON (similar to JSON format).
A simple MongoDB document Structure:

```
{
 title: 'Geeksforgeeks',
 by: 'Harshit Gupta',
 url: 'https://www.geeksforgeeks.org',
 type: 'NoSQL'
}
```

SQL databases store data in tabular format. This data is stored in a predefined data model which is not very much flexible for today's real-world highly growing applications. Modern applications are more networked, social and interactive than ever. Applications are storing more and more data and are accessing it at higher rates.
Relational Database Management System(RDBMS) is not the correct choice when it comes to handling big data by the virtue of their design since they are not horizontally scalable. If the database runs on a single server, then it will reach a scaling limit. NoSQL databases are more scalable and provide superior performance. MongoDB is such a NoSQL database that scales by adding more and more servers and increases productivity with its flexible document model.

Features of MongoDB:

- Document Oriented: MongoDB stores the main subject in the minimal number of documents and not by breaking it up into multiple relational structures like RDBMS. For example, it stores all the information of a computer in a single document called Computer and not in distinct relational structures like CPU, RAM, Hard disk, etc.
- Indexing: Without indexing, a database would have to scan every document of a collection to select those that match the query which would be inefficient. So, for efficient searching Indexing is a must and MongoDB uses it to process huge volumes of data in very less time.
- Scalability: MongoDB scales horizontally using sharding (partitioning data across various servers). Data is partitioned into data chunks using the shard key, and these data chunks are evenly distributed across shards that reside across many physical servers. Also, new machines can be added to a running database.
- Replication and High Availability: MongoDB increases the data availability with multiple copies of data on different servers. By providing redundancy, it protects the database from hardware failures. If one server goes down, the data can be retrieved easily from other active servers which also had the data stored on them.
- Aggregation: Aggregation operations process data records and return the computed results. It is similar to the GROUPBY clause in SQL. A few aggregation expressions are sum, avg, min, max, etc

Where do we use MongoDB?

MongoDB is preferred over RDBMS in the following scenarios:

- Big Data: If you 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 your database.
- 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.
- 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.

Language Support by MongoDB:

MongoDB currently provides official driver support for all popular programming languages like C, C++, Rust, C#, Java, Node.js, Perl, PHP, Python, Ruby, Scala, Go, and Erlang.

Installing MongoDB:

Just go to http://www.mongodb.org/downloads and select your operating system out of Windows, Linux, Mac OS X and Solaris. A detailed explanation about the installation of MongoDB is given on their site.
For Windows, a few options for the 64-bit operating systems drops down. When you're running on Windows 7, 8 or newer versions, select Windows 64-bit 2008 R2+. When you're using Windows XP or Vista then select Windows 64-bit 2008 R2+ legacy.

CONCLUSION - In this way, We studied the MongoDB installation with commands.

PRACTICAL PRINT 1-

Steps of installing MongoDB:

Installing MongoDB

1) Download the MongoDB package suitable for your OS. Download the msi/zip file compatible with your OS and install the application.
2) Open your CMD and type **mongod –version** to verify your installation. If you get an output similar to the result in the picture below, everything is perfect.

```
C:\windows\system32>mongod --version
db version v4.4.1
Build Info: {
    "version": "4.4.1",
    "gitVersion": "ad91a93a5a31e175f5cbf8c69561e788bbc55ce1",
    "modules": [],
    "allocator": "tcmalloc",
    "environment": {
        "distmod": "windows",
        "distarch": "x86_64",
        "target_arch": "x86_64"
    }
}
```

Accessing the Mongo shell

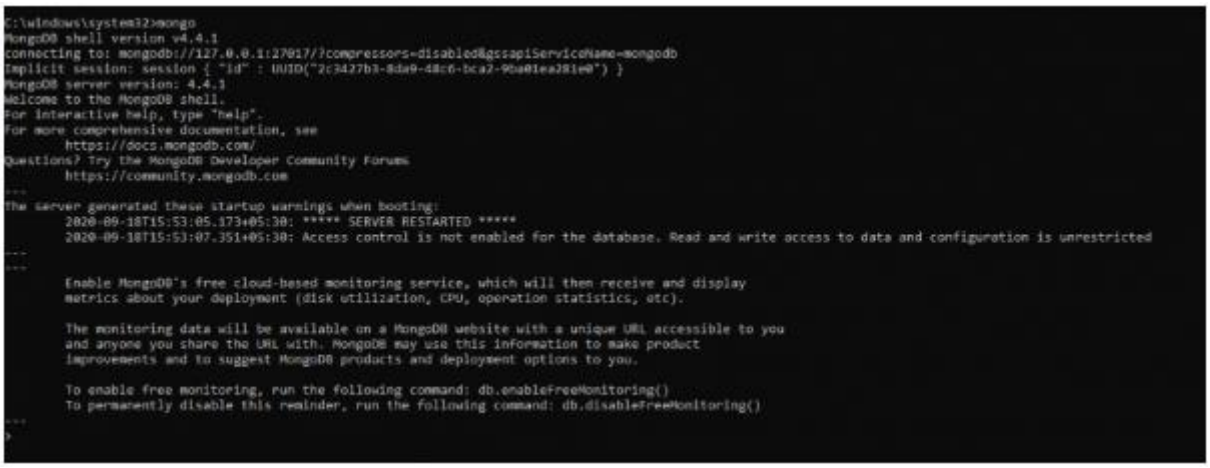
When you install MongoDB, a CLI tool called Mongo shell will be installed along with it. You can use it to give commands to the MySQL server. In order to get access to the mongo shell, you have to start the server with the following command:

```
net start MongoDB
```

```
C:\windows\system32>net start MongoDB
The MongoDB Server (MongoDB) service is starting...
The MongoDB Server (MongoDB) service was started successfully.


C:\windows\system32>
```

Next, we should run the MongoDB shell. To do that, type the command **mongo** in the CMD.



Now you are in the MongoDB shell, where you can execute commands to:

- Create databases
- Insert data
- Edit and delete data from databases
- Issue administrative commands

Creating a Mongo database

MongoDB has a keyword called 'use,' which is used to switch to a specified database. For example, if you want to connect with an existing database named Employee, you can use the command 'use Employee' to connect with it. However, if there is no database as Employee, the command 'use' will create it and connects you to it. Let's use the 'use' command to create our database.

```
use Company
```

When you use the command, you will see the following output:

```
> use company
switched to db company
>
```

Mongo has created the company database and connected you to it. You can use the **db** command to confirm, which will display the currently connected database.

```
> db
company
>
```

You might want to see all the existing databases before creating a new database. You can do it with the **show dbs** command, and it will return the following result:

```
> show dbs
admin   0.000GB
config  0.000GB
local   0.000GB
>
```

Installing MongoDB creates three default databases:

- Admin
- Config
- Local

Surprisingly, the database we just created is not listed. This is because MongoDB doesn't truly create the database until we save values to it.

One of the critical features in NoSQL databases like MongoDB is that they are schema-less—there's no schema. That means you don't specify a structure for the databases/tables as you do in SQL. Instead of tables and rows in SQL, MongoDB has collections and documents. So, let's see how you work with collections in MongoDB.

Using collections in MongoDB

In SQL, you have tables that have a well-defined structure with columns and specified data types. You insert data into tables as records. Every record you insert should have values (including null values) for every column defined on the table.

In contrast, NoSQL has collections instead of tables. These collections don't have any structure, and you insert data, as documents, into the collection. That means that one document can be completely different from another document. Let's create a collection called Employee and add a document to it.

Before you could create a collection, remember that you add data to documents in JSON format. So, the command to insert one document to our database will look like this:

```
db.Employee.insert(
        {
                "Employeename" : "Chris",
                "EmployeeDepartment" : "Sales"
        }
)
```

First, switch to the database to add values using the 'use' command. Then **use db.Employee.insert** command to add a document to the database:

- Db refers to the currently connected database.
- Employee is the newly created collection on the company database.

A significant issue you'll notice in this query is we haven't set a primary key. MongoDB automatically creates a primary key field called _id and sets a default value to it. To see this, let's view the collection we just created. Run the following command to access our collection in JSON format.

```
db.Employee.find().forEach(printjson)
```

```
> db.Employee.find().forEach(printjson)
{
        "_id" : ObjectId("5f64bb2b9acf97fa418e9c00"),
        "Employeename" : "Chris",
        "EmployeeDepartment" : "Sales"
}
{
        "_id" : ObjectId("5f64e6b19acf97fa418e9c01"),
        "Employeename" : "Chris",
        "EmployeeDepartment" : "Sales"
}
>
```

In MongoDB, you cannot set an arbitrary field to the primary key. However, you can change the value of the default primary key. Let's add another document while giving value for _id as well.

```
db.Employee.insert(
        {
                "_id" : 1,
                "EmployeeName" : "Mark",
                "EmployeeDepartment" : "Marketing"
        }
)
```

```
show dbs
```

```
> show dbs
admin    0.000GB
company  0.000GB
config   0.000GB
local    0.000GB
>
```

Removing a database

It's not good to leave this meaningless database on your server. So, let's remove it as it is a good chance for us to learn how to drop a database. To drop any database you want, first, you have to connect with it using the use keyword.

```
use Company
```

Next, type **db.dropDatabase()** to remove the database from your server.

```
> db.dropDatabase()
{ "dropped" : "company", "ok" : 1 }
>
```

Run the **show dbs** command again to verify it.

```
> show dbs
admin   0.000GB
config  0.000GB
local   0.000GB
>
```

PROBLEM STATEMENT -

Implement Map-reduce and aggregation, indexing with suitable example in MongoDB.
Demonstrate the following:
- Aggregation framework
- Create and drop different types of indexes and explain () to show the advantage of the indexes.

THEORY -

MongoDB - Indexing

Indexes support the efficient resolution of queries. Without indexes, MongoDB must scan every document of a collection to select those documents that match the query statement. This scan is highly inefficient and require MongoDB to process a large volume of data.

Indexes are special data structures, that store a small portion of the data set in an easy-to-traverse form. The index stores the value of a specific field or set of fields, ordered by the value of the field as specified in the index.

Indexing Use-

Indexes can improve the efficiency of read operations. The Analyze Query Performance tutorial provides an example of the execution statistics of a query with and without an index.

For information on how MongoDB chooses an index to use, see query optimizer.

MongoDB - Map Reduce

As per the MongoDB documentation, Map-reduce is a data processing paradigm for condensing large volumes of data into useful aggregated results. MongoDB uses mapReduce command for map-reduce operations. MapReduce is generally used for processing large data sets.

In MongoDB, the map-reduce operation can write results to a collection or return the results inline. If you write map-reduce output to a collection, you can perform subsequent map-reduce operations on the same input collection that merge replace, merge, or reduce new results with previous results

MongoDB - Map Aggregation

Aggregations operations process data records and return computed results. Aggregation operations group values from multiple documents together, and can perform a variety of operations on the grouped data to return a single result. In SQL count(*) and with group by is an equivalent of MongoDB aggregation.

the aggregation pipeline has enabled us to do a lot with this example, from determining how many documents are in a collection and being able to run complex operations against that collection, to gathering an average across multiple data points and modifying the collection in the database.

CONCLUSION - I n this way, We Implement Map-reduce and aggregation, indexing with suitable example in MongoDB.

PRACTICAL PRINT 2-

1) Indexing COMMAND

The createIndex() Method

To create an index, you need to use createIndex() method of MongoDB.

Syntax

The basic syntax of createIndex() method is as follows().

```
>db.COLLECTION_NAME.createIndex({KEY:1})
```

Here key is the name of the field on which you want to create index and 1 is for ascending order. To create index in descending order you need to use -1.

Example

```
>db.mycol.createIndex({"title":1})
{
        "createdCollectionAutomatically" : false,
        "numIndexesBefore" : 1,
        "numIndexesAfter" : 2,
        "ok" : 1
}
>
```

In createIndex() method you can pass multiple fields, to create index on multiple fields.

```
>db.mycol.createIndex({"title":1,"description":-1})
```

This method also accepts list of options (which are optional). Following is the list −

Parameter	Type	Description
background	Boolean	Builds the index in the background so that building an index does not block other database activities. Specify true to build in the background. The default value is false.
unique	Boolean	Creates a unique index so that the collection will not accept insertion of documents where the index key or keys match an existing value in the index. Specify true to create a unique index. The default value is false.
name	string	The name of the index. If unspecified, MongoDB generates an index name by concatenating the names of the indexed fields and the sort order.
sparse	Boolean	If true, the index only references documents with the specified field. These indexes use less space but behave differently in some situations (particularly sorts). The default value is false.
expireAfterSeconds	integer	Specifies a value, in seconds, as a TTL to control how long MongoDB retains documents in this collection.

weights	document	The weight is a number ranging from 1 to 99,999 and denotes the significance of the field relative to the other indexed fields in terms of the score.
default_language	string	For a text index, the language that determines the list of stop words and the rules for the stemmer and tokenizer. The default value is English.
language_override	string	For a text index, specify the name of the field in the document that contains, the language to override the default language. The default value is language.

The dropIndex() method

You can drop a particular index using the dropIndex() method of MongoDB.

Syntax

The basic syntax of DropIndex() method is as follows().

>db.COLLECTION_NAME.dropIndex({KEY:1})

Here, "key" is the name of the file on which you want to remove an existing index. Instead of the index specification document (above syntax), you can also specify the name of the index directly as:

dropIndex("name_of_the_index")

Example

```
> db.mycol.dropIndex({"title":1})
{
        "ok" : 0,
        "errmsg" : "can't find index with key: { title: 1.0 }",
        "code" : 27,
        "codeName" : "IndexNotFound"
}
```

The dropIndexes() method

This method deletes multiple (specified) indexes on a collection.

Syntax

The basic syntax of DropIndexes() method is as follows() −

>db.COLLECTION_NAME.dropIndexes()

Example

Assume we have created 2 indexes in the named mycol collection as shown below −

```
> db.mycol.createIndex({"title":1,"description":-1})
```

Following example removes the above created indexes of mycol −

```
>db.mycol.dropIndexes({"title":1,"description":-1})
{ "nIndexesWas" : 2, "ok" : 1 }
>
```

The getIndexes() method

This method returns the description of all the indexes int the collection.

Syntax

Following is the basic syntax od the getIndexes() method −

db.COLLECTION_NAME.getIndexes()

Example

Assume we have created 2 indexes in the named mycol collection as shown below −

```
> db.mycol.createIndex({"title":1,"description":-1})
```

Following example retrieves all the indexes in the collection mycol −

```
> db.mycol.getIndexes()
[
        {
                "v" : 2,
                "key" : {
                        "_id" : 1
                },
                "name" : "_id_",
                "ns" : "test.mycol"
        },
        {
                "v" : 2,
                "key" : {
                        "title" : 1,
                        "description" : -1
                },
                "name" : "title_1_description_-1",
                "ns" : "test.mycol"
        }
]
```

Following is the syntax of the basic mapReduce command −

```
>db.collection.mapReduce(
   function() {emit(key,value);},  //map function
   function(key,values) {return reduceFunction}, {   //reduce function
      out: collection,
      query: document,
      sort: document,
      limit: number
   }
)
```

The map-reduce function first queries the collection, then maps the result documents to emit key-value pairs, which is then reduced based on the keys that have multiple values.

In the above syntax −

- map is a javascript function that maps a value with a key and emits a key-value pair
- reduce is a javascript function that reduces or groups all the documents having the same key
- out specifies the location of the map-reduce query result
- query specifies the optional selection criteria for selecting documents
- sort specifies the optional sort criteria
- limit specifies the optional maximum number of documents to be returned

Using MapReduce

Consider the following document structure storing user posts. The document stores user_name of the user and the status of post.

```
{
   "post_text": "tutorialspoint is an awesome website for tutorials",
   "user_name": "mark",
   "status":"active"
}
```

Now, we will use a mapReduce function on our posts collection to select all the active posts, group them on the basis of user_name and then count the number of posts by each user using the following code −

```
>db.posts.mapReduce(
   function() { emit(this.user_id,1); },

   function(key, values) {return Array.sum(values)}, {
      query:{status:"active"},
      out:"post_total"
   }
)
```

The above mapReduce query outputs the following result −

```
{
  "result" : "post_total",
  "timeMillis" : 9,
  "counts" : {
    "input" : 4,
    "emit" : 4,
    "reduce" : 2,
    "output" : 2
  },
  "ok" : 1,
}
```

The result shows that a total of 4 documents matched the query (status:"active"), the map function emitted 4 documents with key-value pairs and finally the reduce function grouped mapped documents having the same keys into 2.

To see the result of this mapReduce query, use the find operator −

```
>db.posts.mapReduce(
  function() { emit(this.user_id,1); },
  function(key, values) {return Array.sum(values)}, {
    query:{status:"active"},
    out:"post_total"
  }

).find()
```

The above query gives the following result which indicates that both users tom and mark have two posts in active states −

```
{ "_id" : "tom", "value" : 2 }
{ "_id" : "mark", "value" : 2 }
```

In a similar manner, MapReduce queries can be used to construct large complex aggregation queries. The use of custom Javascript functions make use of MapReduce which is very flexible and powerful.

3) Aggregate COMMAND

For the aggregation in MongoDB, you should use aggregate() method.

Basic syntax of aggregate() method is as follows −

>db.COLLECTION_NAME.aggregate(AGGREGATE_OPERATION)

Example

In the collection you have the following data −

```
{
  _id: ObjectId(7df78ad8902c)
  title: 'MongoDB Overview',
  description: 'MongoDB is no sql database',
  by_user: 'tutorials point',
  url: 'http://www.tutorialspoint.com',
  tags: ['mongodb', 'database', 'NoSQL'],
  likes: 100
},
{
  _id: ObjectId(7df78ad8902d)
  title: 'NoSQL Overview',
  description: 'No sql database is very fast',
  by_user: 'tutorials point',
  url: 'http://www.tutorialspoint.com',
  tags: ['mongodb', 'database', 'NoSQL'],
  likes: 10
},
{
  _id: ObjectId(7df78ad8902e)
  title: 'Neo4j Overview',
  description: 'Neo4j is no sql database',
  by_user: 'Neo4j',
  url: 'http://www.neo4j.com',
  tags: ['neo4j', 'database', 'NoSQL'],
  likes: 750
},
```

Now from the above collection, if you want to display a list stating how many tutorials are written by each user, then you will use the following aggregate() method −

```
> db.mycol.aggregate([{$group : {_id : "$by_user", num_tutorial : {$sum : 1}}}])
{ "_id" : "tutorials point", "num_tutorial" : 2 }
{ "_id" : "Neo4j", "num_tutorial" : 1 }
>
```

Sql equivalent query for the above use case will be select by_user, count(*) from mycol group by by_user.