ASSIGNMENT NO: 01 NoSQL Database

AIM: Study of Open Source Large Databases: MongoDB, Write Difference between SQL and NoSQL Databases. Installation Steps for MongoDB, Features of MongoDB and Basic CRUD operation on MongoDB, with suitable example.

Theory:
MongoDB is an open-source document database, and leading NoSQL database. MongoDB is written in C++.

Key Features
Since MongoDB offers a Document oriented storage, it is simple and easily programmable.
You can set an index on any attribute of a MongoDB record as follows-

```
("FirstName":"Sameer", "Address":"Gandhi Road")
```

With respect to which, a record can be sort quickly and ordered. You can set mirror across local as well as wide area networks, which makes it easily scalable. If load increases (more storage space, more processing power), it can be distributed to other nodes across computer networks. This is called as sharding. MongoDB supports rich query to fetch data from the database.
MongoDB supports replacing an entire document (database) or some specific fields with its update() command.
MongoDB supports Map/Reduce framework for batch processing of data and aggregation operation.

Here is in brief of how Map/Reduce works:

Map: A master node takes an input. Splits it into smaller sections. Sends it to the associated nodes. These nodes may perform the same operation in turn to send those smaller sections of input to other nodes. It processes the problem (taken as input) and sends it back to the Master Node.

Reduce: The master node aggregates those results to find the output.
GridFS specification of MongoDB supports storage of very large files.
MongoDB supports various programming languages like C, C# and .NET, C++, Erlang, Haskell, Java, Javascript, Perl, PHP, Python, Ruby, Scala (via Casbah).
It supports Server-side JavaScript execution. Which allows a developer to use a single programming language for both client and server side code.MongoDB is easily installable.

History
Development of MongoDB began in October 2007 by 10gen. The first public release was in February 2009.

Definitions
MongoDB is a cross-platform, document oriented database that provides, high performance, high availability, and easy scalability. MongoDB works on concept of collection and document.

Database: Database is a physical container for collections. Each database gets its own set of files on the file system. A single MongoDB server typically has multiple databases.

Collection: Collection is a group of MongoDB documents. It is the equivalent of an RDBMS table. A collection exists within a single database. Collections do not enforce a schema. Documents within a collection can have different fields. Typically, all documents in a collection are of similar or related purpose.

Document: A document is a set of key-value pairs. Documents have dynamic schema. Dynamic schema means that documents in the same collection do not need to have the same set of fields or structure, and common fields in a collection's documents may hold different types of data.

Below given table shows the relationship of RDBMS terminology with MongoDB

RDBMS	MongoDB
Database	Database
Table	Collection
Tuple/Row	Document
column	Field
Table Join	Embedded Documents
Primary Key	Primary Key (Default key _id provided by mongodb itself)

Any relational database has a typical schema design that shows number of tables and the relationship between these tables. While in MongoDB there is no concept of relationship

Advantages of MongoDB over RDBMS

- Schema less: MongoDB is document database in which one collection holds different different documents. Number of fields, content and size of the document can be differ from one document to another.
- Structure of a single object is clear
- No complex joins
- Deep query-ability. MongoDB supports dynamic queries on documents using a document-based query language that's nearly as powerful as SQL
- Tuning
- Ease of scale-out: MongoDB is easy to scale
- Conversion / mapping of application objects to database objects not needed
- Uses internal memory for storing the (windowed) working set, enabling faster access of data
- Why should we use MongoDB?
- Document Oriented Storage : Data is stored in the form of JSON style documents
- Index on any attribute
- Replication & High Availability

- Auto-Sharding
- Rich Queries
- Fast In-Place Updates
- Professional Support By MongoDB
- Where should we use MongoDB?
- Big Data
- Content Management and Delivery
- Mobile and Social Infrastructure
- User Data Management

Installing and running the MongoDB server
MongoDB can be downloaded from the following webpage: https://www.mongodb.org/downloads
Put the downloaded file wherever you prefer, open a terminal, go to the directory where you put the downloaded file, then type the following commands

```
tar -xzf mongodb-linux-x86_64-2.6.5.tgz
cd mongodb-linux-x86_64-2.6.5
sudo mkdir –p /data/db
sudo ./bin/mongod
```

The last command will launch the server, and MongoDB is ready to accept connection on port 27017.

The use Command : MongoDB **use DATABASE_NAME** is used to create database. The command will create a new database, if it doesn't exist otherwise it will return the existing database.

Syntax: Basic syntax of **use DATABASE** statement is as follows:

```
> use DATABASE_NAME
```

Example: If you want to create a database with name **<mydb>**, then **use DATABASE**statement would be as follows:

```
>use mydb

switched to db mydb
```

To check your currently selected database use the command **db**

```
>db
```

```
mydb
```

The dropDatabase() Method: MongoDB **db.dropDatabase()** command is used to drop a existing database.

Syntax: Basic syntax of **dropDatabase()** command is as follows:

```
>db.dropDatabase()
```

This will delete the selected database. If you have not selected any database, then it will delete default 'test' database.

Example: First, check the list available databases by using the command **show dbs**

```
>show dbs

local     0.78125GB
mydb      0.23012GB
test      0.23012GB
```

If you want to delete new database **<mydb>**, then **dropDatabase()** command would be as follows:

```
>use mydb

switched to db mydb

>db.dropDatabase()

{ "dropped" : "mydb", "ok" : 1 }
```

The createCollection() Method : MongoDB **db.createCollection(name, options)**is used to create collection.

Syntax: Basic syntax of **createCollection()** command is as follows

```
>db.createCollection(name, options)
```

In the command, **name** is name of collection to be created. **Options** is a document and used to specify configuration of collection

Examples:

>use test

switched to db test

>db.createCollection("mycollection")

{ "ok" : 1 }

The drop() Method: MongoDB's **db.collection.drop()** is used to drop a collection from the database.

Syntax: Basic syntax of **drop()** command is as follows

db.COLLECTION_NAME.drop()

Example: *> db.mycollection.drop()*

true

The insert() Method: To insert data into MongoDB collection, you need to use MongoDB's **insert()** or **save()**method.

Syntax: Basic syntax of **insert()** command is as follows:

>db.COLLECTION_NAME.insert(document)

Example:

>db.mycol.insert({_id: ObjectId(7df78ad8902c), title: 'MongoDB Overview', description: 'MongoDB is no sql database', by: 'point', url: 'http://www.MONGO.com', tags: ['mongodb', 'database', 'NoSQL'], likes: 100 })

Here **mycol** is our collection name, as created in previous tutorial. If the collection doesn't exist in the database, then MongoDB will create this collection and then insert document into it.

MongoDB's **update()** and **save()** methods are used to update document into a collection. **The update () method update values in the existing document while the save() method replaces the existing document with the document passed in save() method.**

MongoDB Update() method: The update() method updates values in the existing document.

Syntax: Basic syntax of **update()** method is as follows

>db.COLLECTION_NAME.update(SELECTIOIN_CRITERIA, UPDATED_DATA)

Following example will set the new title 'New MongoDB Tutorial' of the documents whose title is 'MongoDB Overview'

>db.mycol.update({'title':'MongoDB Overview'},{$set:{'title':'New MongoDB Tutorial'}})

MongoDB Save() Method: The **save()** method replaces the existing document with the new document passed in save() method

Syntax: Basic syntax of mongodb **save()** method is shown below:

>db.COLLECTION_NAME.save({_id:ObjectId(),NEW_DATA})

Example: Following example will replace the document with the _id '5983548781331adf45ec7'

>db.mycol.save({ "_id" : ObjectId(5983548781331adf45ec7), "title":"Tutorial New Topic", "by":"Tutorial" })

The remove() Method: MongoDB's remove() method is used to remove document from the collection. remove() method accepts two parameters. One is deletion criteria and second is justOne flag.

1. **deletion criteria :** (Optional) deletion criteria according to documents will be removed.
2. **justOne :** (Optional) if set to true or 1, then remove only one document.

Syntax: Basic syntax of remove() method is as follows

>db.COLLECTION_NAME.remove(DELLETION_CRITTERIA)

Following example will remove all the documents whose title is 'MongoDB Overview'

>db.mycol.remove({'title':'MongoDB Overview'})

Remove only one : If there are multiple records and you want to delete only first record, then set justOne parameter in remove() method

>db.COLLECTION_NAME.remove(DELETION_CRITERIA,1)

Remove All documents : If you don't specify deletion criteria, then mongodb will delete whole documents from the collection. This is equivalent of SQL's truncate command.

>db.mycol.remove()

The find() Method : To query data from MongoDB collection, you need to use MongoDB's find() method.

Syntax: Basic syntax of find() method is as follows

>db.COLLECTION_NAME.find()

AND in MongoDB:

Syntax: In the find() method if you pass multiple keys by separating them by ',' then MongoDB treats it AND condition. Basic syntax of AND is shown below:

>db.mycol.find({key1:value1, key2:value2}).pretty()

OR in MongoDB :

Syntax: To query documents based on the OR condition, you need to use $or keyword. Basic syntax of OR is shown below:

> db.mycol.find({ $or: [{key1: value1}, {key2:value2}] })

Conclusion: Hence we have studied basic operation commands of MongoDB in two tier architecture

********************** QUERIES *********************

AIM: Implement database with suitable example using MongoDB and implement all basic operations.

```
 > show dbs
admin  (empty)
local  0.078GB
te_a   0.078GB



> use te_a
switched to db te_a



> db.expt5.insert({"RNO" : 1, "NAME" : { "FIRST" : "SANDESH", "LNAME" :
"GANGWAL" }, "ADDRESS" : { "CITY" : "PUNE", "DISTRICT" : "PUNE", "PIN" :
345678 }, "CONTACTNO" :[1234567,456788076] });


WriteResult({ "nInserted" : 1 })




> db.expt5.insert({"RNO" : 2, "NAME" : { "FIRST" : "ISHAAN", "LNAME" :
"KOTHARI" }, "ADDRESS" : { "CITY" : "AMRAVATI", "DISTRICT" : "AMRAVATI",
"PIN" : 444605 }, "CONTACTNO" : 789798797});


WriteResult({ "nInserted" : 1 })


 > db.expt5.insert({"RNO" : 3, "NAME" : { "FIRST" : "AKASH", "LNAME" :
"SUMEDH" }, "ADDRESS" : { "CITY" : "PUNE", "DISTRICT" : "PUNE", "PIN" :
444605 }, "CONTACTNO" :[34343444,789798797,356565665] ,   "EMAIL" :
"jfsdkfj@dkfj.com"});


WriteResult({ "nInserted" : 1 })


 > show collections
expt5
system.indexes



> db.expt5.find();


{ "_id" : ObjectId("541e3d71313dc145de686df5"), "RNO" : 2, "NAME" : {
"FIRST" : "ISHAAN", "LNAME" : "KOTHARI" }, "ADDRESS" : { "CITY" :
```

```
"AMRAVATI", "DISTRICT" : "AMRAVATI", "PIN" : 444605 }, "CONTACTNO" :
789798797 }

{ "_id" : ObjectId("541e3dcb313dc145de686df6"), "RNO" : 1, "NAME" : {
"FIRST" : "SANDESH", "LNAME" : "GANGWAL" }, "ADDRESS" : { "CITY" : "PUNE",
"DISTRICT" : "PUNE", "PIN" : 345678 }, "CONTACTNO" : [ 1234567, 456788076
] }

{ "_id" : ObjectId("541e3e5a313dc145de686df7"), "RNO" : 3, "NAME" : {
"FIRST" : "AKASH", "LNAME" : "SUMEDH" }, "ADDRESS" : { "CITY" : "PUNE",
"DISTRICT" : "PUNE", "PIN" : 444605 }, "CONTACTNO" : [ 34343444,
789798797, 356565665 ],      "EMAIL" : "jfsdkfj@dkfj.com" }

 > db.expt5.find().pretty();

{
      "_id" : ObjectId("541e3d71313dc145de686df5"),
      "RNO" : 2,
      "NAME" : {
            "FIRST" : "ISHAAN",
            "LNAME" : "KOTHARI"
      },
      "ADDRESS" : {
            "CITY" : "AMRAVATI",
            "DISTRICT" : "AMRAVATI",
            "PIN" : 444605
      },
      "CONTACTNO" : 789798797
}
{
      "_id" : ObjectId("541e3dcb313dc145de686df6"),
      "RNO" : 1,
      "NAME" : {
            "FIRST" : "SANDESH",
            "LNAME" : "GANGWAL"
      },
      "ADDRESS" : {
            "CITY" : "PUNE",
      "DISTRICT" : "PUNE",
            "PIN" : 345678
      },
      "CONTACTNO" : [          1234567,
            456788076   ]
}
{
"_id" : ObjectId("541e3e5a313dc145de686df7"),
      "RNO" : 3,
      "NAME" : {
            "FIRST" : "AKASH",
            "LNAME" : "SUMEDH"
      },
      "ADDRESS" : {
            "CITY" : "PUNE",
            "DISTRICT" : "PUNE",
            "PIN" : 444605
      },
      "CONTACTNO" : [
            34343444,
            789798797,
            356565665
      ]
      "EMAIL" : "jfsdkfj@dkfj.com"
}



>
db.expt5.update({"NAME.LNAME":"SUMEDH"},{$set:{"NAME.LNAME":"KOTHARI"}});
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })



>
db.expt5.update({"ADDRESS.CITY":"PUNE"},{$set:{"CONTACTNO":8888888888}});
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })



>
db.expt5.update({"ADDRESS.CITY":"PUNE"},{$set:{"CONTACTNO":8888888888}},{m
ulti:true});
WriteResult({ "nMatched" : 2, "nUpserted" : 0, "nModified" : 1 })



>
db.expt5.update({"RNO":{$gt:5}},{$set:{"CONTACTNO":8888888888}},{multi:tru
e});

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

 >
db.expt5.update({"RNO":{$lt:5}},{$set:{"CONTACTNO":8888888888}},{multi:tru
e});

WriteResult({ "nMatched" : 3, "nUpserted" : 0, "nModified" : 1 })



> db.expt5.find().pretty();
{
      "_id" : ObjectId("541cfdb18a5439b34ba0b41a"),
      "RNO" : 1,
      "NAME" : {
            "FIRST" : "SANDESH",
            "LNAME" : "GANGWAL"
      },
      "ADDRESS" : {
```

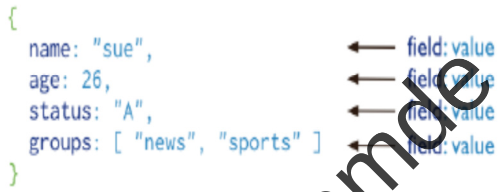
```
                "CITY" : "PUNE",
                "DISTRICT" : "PUNE",
                "PIN" : 345678
        },
        "CONTACTNO" : 8888888888
}
{
        "_id" : ObjectId("541cfdec8a5439b34ba0b41b"),
        "RNO" : 2,
        "NAME" : {
                "FIRST" : "ISHAAN",
                "LNAME" : "KOTHARI"
        },
        "ADDRESS" : {
                "CITY" : "AMRAVATI",
                "DISTRICT" : "AMRAVATI",
                "PIN" : 444605
        },
        "CONTACTNO" : 8888888888
}
{
        "_id" : ObjectId("541cff058a5439b34ba0b41d"),
        "RNO" : 3,
        "NAME" : {
                "FIRST" : "AKASH",
                "LNAME" : "KOTHARI"
        },
        "ADDRESS" : {
                "CITY" : "PUNE",
                "DISTRICT" : "PUNE",
                "PIN" : 444605
        },
        "CONTACTNO" : 8888888888,
        "EMAIL" : "jfsdkfj@dkfj.com"
}

> db.expt5.remove({"ADDRESS.CITY":"AMRAVATI"});
WriteResult({ "nRemoved" : 1 })

> db.expt5.find().pretty();
{
        "_id" : ObjectId("541cfdb18a5439b34ba0b41a"),
        "RNO" : 1,
        "NAME" : {
                "FIRST" : "SANDESH",
                "LNAME" : "GANGWAL"
        },
        "ADDRESS" : {
                "CITY" : "PUNE",
                "DISTRICT" : "PUNE",
                "PIN" : 345678
        },
        "CONTACTNO" : 8888888888
}
{
        "_id" : ObjectId("541cff058a5439b34ba0b41d"),
        "RNO" : 3,
        "NAME" : {
              "FIRST" : "AKASH",
              "LNAME" : "KOTHARI"
        },
        "ADDRESS" : {
              "CITY" : "PUNE",
              "DISTRICT" : "PUNE",
              "PIN" : 444605
        },
        "CONTACTNO" : 8888888888,
        "EMAIL" : "jfsdkfj@dkfj.com"
}


> db.expt5.remove({"ADDRESS.CITY":"PUNE"},1);

WriteResult({ "nRemoved" : 1 })

 > db.expt5.drop();

true

> db.dropDatabase();

{ "dropped" : "te_a", "ok" : 1 }
```

ASSIGNMENT NO: 02 NoSQL Database

AIM: Perform the basic CRUD operation on MongoDB, with suitable example using SAVE() method, use logical operators to solve queries.

Theory:

In computer programming, create, read, update and delete (as an acronym CRUD) are the four basic functions of persistent storage, It is also sometimes used to describe user interface conventions that facilitate viewing, searching, and changing information..


```
{
   name: "sue",                          ← field: value
   age: 26,                              ← field: value
   status: "A",                          ← field: value
   groups: [ "news", "sports" ]          ← field: value
}
```

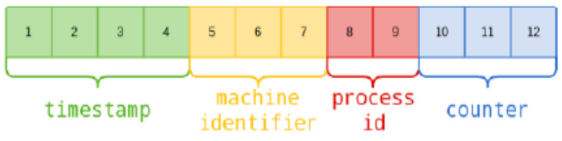
MongoDB stores data in the form of JSON-like value pair documents; these documents are analogous to Structures/Classes in programming languages that associate keys with values (e.g. dictionaries, hashes, maps, and associative arrays).

Formally, MongoDB documents are BSON (binary representation of JSON) documents with additional type information. In BSON documents, the value of a field can be any of the BSON data types, including other documents, arrays, and arrays of documents.

To avoid these bottlenecks MongoDB uses *ObjectId* as a default , which uses 12 bytes of storage. Now this 12 bytes are very interestingly divided into 4 sub parts to ensure that you will always get unique _id value in any case → even though items may be on hundreds of computers in many data clusters.

- _id generated on two different machines.
- _id generated on the same machine but by two different processes.
- _id generated on the same machine ,by the same process but in same second.

The following graphic shows the Bytes Distribution for ObjectID:



MongoDB's **update()** and **save()** methods are used to update document into a collection. **The update () method update values in the existing document while the save()**

method replaces the existing document with the document passed in save() method.

MongoDB Update() method: The update() method updates values in the existing document.

Syntax: Basic syntax of **update()** method is as follows

>db.COLLECTION_NAME.update(SELECTIOIN_CRITERIA, UPDATED_DATA)

Following example will set the new title 'New MongoDB Tutorial' of the documents whose title is 'MongoDB Overview'

>db.mycol.update({'title':'MongoDB Overview'},{$set:{'title':'New MongoDB Tutorial'}})

MongoDB Save() Method: The **save()** method replaces the existing document with the new document passed in save() method

Syntax: Basic syntax of mongodb **save()** method is shown below:

>db.COLLECTION_NAME.save({_id:ObjectId(),NEW_DATA})

Example: Following example will replace the document with the _id '5983548781331adf45ec7'

>db.mycol.save({ "_id" : ObjectId(5983548781331adf45ec7), "title":"Tutorial New Topic", "by":"Tutorial" })

AND in MongoDB:

Syntax: In the find() method if you pass multiple keys by separating them by ',' then MongoDB treats it AND condition. Basic syntax of AND is shown below:

>db.mycol.find({key1:value1, key2:value2}).pretty()

OR in MongoDB :

Syntax: To query documents based on the OR condition, you need to use $or keyword. Basic syntax of OR is shown below:

> db.mycol.find({ $or: [{key1: value1}, {key2:value2}] })

$not Operator

In the Mongo universe, the **$not** operator performs a logical *NOT* operation on the specified <operator-expression> and gets the documents from a collection that:

- Do not match the <operator-expression>
- Do not contain the field given by the <operator-expression>

Here is what the query syntax will look like.

Syntax

```
1 db.collection_name.find( { field_name: { $not: { <operator-expression> } } } )
```

$nor Operator

In the Mongo universe, the **$nor** operator performs on an array of two or more expressions (e.g., <Expression 1>, <Expression 2> etc.). This operator is similar to a logical *NOR* operation and gets the documents from a collection that fail all the expressions in an array. Here is what the query syntax will look like.

Syntax

```
1 > db.collection_name.find( { $nor: [ {  }, {  } , ... , {  } ] } )
```

Try (compression) operator:

$lt,$lte,$gt,$gte, $project,$match,$limit,$skip

Conclusion

In this, we learned about the different logical query operators of the Mongo database.

ASSIGNMENT NO: 03 NoSQL Database

AIM: Aggregation and indexing with suitable example using MongoDB.

Theory:

Indexes: 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 the mongod 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 index.

The ensureIndex() Method: To create an index you need to use ensureIndex() method of mongodb.

Syntax: Basic syntax of ensureIndex() method is as follows()

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

Here key is the name of filed 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.ensureIndex({“title”:1})
```

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

```
>db.mycol.ensureIndex({“title”:1,”description”:-1})
```

explain() method : This method provides information on the query plan. The query plan is the plan the server uses to find the matches for a query. This information may be useful when optimizing a query. The explain() method returns a document that describes the process used to return the query results. The explain() method has the following form:

```
db.collection.find().explain()
```

Aggregations: 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 aggregate() Method: For the aggregation in mongodb, aggregate() method is used.

Syntax: Basic syntax of aggregate() method is as follows

```
>db.COLLECTION_NAME.aggregate(AGGREGATE_OPERATION)
```

Now from the above collection if it is required to display a list that how many tutorials are written by each user then aggregate() method is used as shown below:

> db.mycol.aggregate([{$group : {_id : “$by_user”, num_tutorial : {$sum : 1}}}])

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

There is a list available aggregation expressions.

1. **$sum** Sums up the defined value from all documents in the collection.

db.mycol.aggregate([{$group : {_id : “$by_user”, num_tutorial : {$sum : “$likes”}}}])

2. **$avg** Calculates the average of all given values from all documents in the collection. mycol.aggregate([{$group : {_id : “$by_user”, num_tutorial : {$avg : “$likes”}}}])

3. **$min** Gets the minimum of the corresponding values from all documents in the collection. mycol.aggregate([{$group : {_id : “$by_user”, num_tutorial : {$min : “$likes”}}}])

4. **$max** Gets the maximum of the corresponding values from all documents in the collection. mycol.aggregate([{$group : {_id : “$by_user”, num_tutorial : {$max : “$likes”}}}])

5. **$push** Inserts the value to an array in the resulting document.mycol.aggregate([{$group : {_id : “$by_user”, url : {$push: “$url”}}}])

6. **$addToSet** Inserts the value to an array in the resulting document but does not create duplicates. mycol.aggregate([{$group : {_id : "$by_user", url : {$addToSet : "$url"}}}])

7. **$first** Gets the first document from the source documents according to the grouping. Typically this makes only sense together with some previously applied "$sort"-stage. mycol.aggregate([{$group : {_id : "$by_user", first_url : {$first : "$url"}}}])

8. **$last** Gets the last document from the source documents according to the grouping. Typically this makes only sense together with some previously applied "$sort"-stage. mycol.aggregate([{$group : {_id : "$by_user", last_url : {$last : "$url"}}}])

Conclusion : This assignment explains details about use of indexing and aggregation in MongoDB.

************************** QUERIES ****************************

```
AIM : Aggregation and indexing with suitable example using MongoDB.

> show collections
bank
expt5
sales
system.indexes



> db.expt5.ensureIndex({"RNO":1},{"UNIQUE":true});
{
      "createdCollectionAutomatically" : false,
      "numIndexesBefore" : 1,
      "numIndexesAfter" : 2,
      "ok" : 1
}

> db.expt5.find().explain();
{
      "cursor" : "BasicCursor",
      "isMultiKey" : false,
      "n" : 3,
      "nscannedObjects" : 3,
      "nscanned" : 3,
      "nscannedObjectsAllPlans" : 3,
      "nscannedAllPlans" : 3,
      "scanAndOrder" : false,
      "indexOnly" : false,
      "nYields" : 0,
      "nChunkSkips" : 0,
      "millis" : 0,
      "server" : "localhost.localdomain:27017",
      "filterSet" : false
}



> db.expt5.insert({"RNO" : 1, "NAME" : { "FIRST" : "SANDESH", "LNAME" : "GANGWAL"
},
"ADDRESS" : { "CITY" : "PUNE", "DISTRICT" : "PUNE", "PIN" : 345678 }, "CONTACTNO"
:[1234567,456788076] });

WriteResult({
      "nInserted" : 0,
      "writeError" : {
            "code" : 11000,
            "errmsg" : "insertDocument :: caused by :: 11000 E11000 duplicate key
error index: te_a.expt5.$RNO_1  dup key: { : 1.0 }"
      }
})

> db.bank.insert({"CUSTID":"B1","ACCTYPE":"SB","BRANCH":"PUNE","BALANCE":2300});
WriteResult({ "nInserted" : 1 })

> db.bank.insert({"CUSTID":"A1","ACCTYPE":"SB","BRANCH":"PUNE","BALANCE":8900});
WriteResult({ "nInserted" : 1 })
```

```
> db.bank.insert({"CUSTID":"A1","ACCTYPE":"CU","BRANCH":"PUNE","BALANCE":0900});
 WriteResult({ "nInserted" : 1 })

> db.bank.insert({"CUSTID":"B1","ACCTYPE":"CU","BRANCH":"PUNE","BALANCE":2900});
WriteResult({ "nInserted" : 1 })

> db.bank.insert({"CUSTID":"B1","ACCTYPE":"CU","BRANCH":"WAGHOLI","BALANCE":2900})
;
WriteResult({ "nInserted" : 1 })

> db.bank.insert({"CUSTID":"A1","ACCTYPE":"CU","BRANCH":"WAGHOLI","BALANCE":5600})
;
 WriteResult({ "nInserted" : 1 })


> db.bank.find();
{ "_id" : ObjectId("541e535f313dc145de686dfd"), "CUSTID" : "B1", "ACCTYPE" : "SB",
"BRANCH" : "PUNE", "BALANCE" : 2300 }
{ "_id" : ObjectId("541e536d313dc145de686dfe"), "CUSTID" : "A1", "ACCTYPE" : "SB",
"BRANCH" : "PUNE", "BALANCE" : 8900 }
{ "_id" : ObjectId("541e5379313dc145de686dff"), "CUSTID" : "A1", "ACCTYPE" : "CU",
"BRANCH" : "PUNE", "BALANCE" : 900 }
{ "_id" : ObjectId("541e5386313dc145de686e00"), "CUSTID" : "B1", "ACCTYPE" : "CU",
"BRANCH" : "PUNE", "BALANCE" : 2900 }
{ "_id" : ObjectId("541e5398313dc145de686e01"), "CUSTID" : "B1", "ACCTYPE" : "CU",
"BRANCH" : "WAGHOLI", "BALANCE" : 2900 }
{ "_id" : ObjectId("541e53a3313dc145de686e02"), "CUSTID" : "A1", "ACCTYPE" : "CU",
"BRANCH" : "WAGHOLI", "BALANCE" : 5600 }

> db.bank.ensureIndex({"CUSTID":1,"ACCTYPE":1,"BRANCH":1},{"unique":1});
{
      "createdCollectionAutomatically" : false,
      "numIndexesBefore" : 1,
      "numIndexesAfter" : 2,
      "ok" : 1
}

>
db.bank.insert({"CUSTID":"B1","ACCTYPE":"CU","BRANCH":"WAGHOLI","BALANCE":2900});
WriteResult({
      "nInserted" : 0,
      "writeError" : {
            "code" : 11000,
            "errmsg" : "insertDocument :: caused by :: 11000 E11000 duplicate key
error index: te_a.bank.$CUSTID_1_ACCTYPE_1_BRANCH_1  dup key: { : \"B1\", :
\"CU\", : \"WAGHOLI\" }"
      }
})

> db.bank.aggregate([{$group : {_id : "$CUSTID", count : {$sum : 1}}}]);
{ "_id" : "A1", "count" : 3 }
{ "_id" : "B1", "count" : 3 }

> db.bank.aggregate([{$group : {_id : "$CUSTID", BALANCE : {$sum :
"$BALANCE"}}}]);
{ "_id" : "A1", "BALANCE" : 15400 }
{ "_id" : "B1", "BALANCE" : 8100 }

> db.bank.aggregate([{$group : {_id : "$BRANCH", BALANCE : {$avg :
"$BALANCE"}}}]);

{ "_id" : "WAGHOLI", "BALANCE" : 4250 }
{ "_id" : "PUNE", "BALANCE" : 3750 }

> db.bank.aggregate([{$group : {_id : 1, BALANCE : {$avg : "$BALANCE"}}}]);
{ "_id" : 1, "BALANCE" : 3916.6666666666665 }

> db.bank.aggregate([{$group : {_id : 1, BALANCE : {$sum : "$BALANCE"}}}]);
{ "_id" : 1, "BALANCE" : 23500 }

> db.bank.aggregate([{$group : {_id : 1, count : {$sum : 1}}}]);
{ "_id" : 1, "count" : 6 }

> db.bank.aggregate([{$group : {_id : 1, count : {$min : "$BALANCE"}}}]);
{ "_id" : 1, "count" : 900 }

> db.bank.aggregate([{$group : {_id : "$ACCTYPE", BALANCE : {$max :
"$BALANCE"}}}]);
 { "_id" : "CU", "BALANCE" : 5600 }
{ "_id" : "SB", "BALANCE" : 8900 }
```

Another example for Aggregation Functions

The Customer entity represents someone making a purchase, so we'll include some data typically found in a Customer entity. A typical customer entity might look like the following:

```
{
 "id": "1",
 "firstName": "Jane",
 "lastName": "Doe",
 "phoneNumber": "999999999",
 "city": "Beverly Hills",
 "state: "CA",
 "zip": 90210
 "email": "Jane.Doe@gmail.com"
}
```

Customer's collection contains any number of customers, each with a similar data format. To begin an aggregation on the customers collection, we call the aggregate function on the customers collection:

> db.customers.aggregate([... aggregation steps go here ...]);

The aggregate function accepts an array of data transformations which are applied to the data in the order they're defined. This makes aggregation a lot like other data flow pipelines: the transformations that are defined first will be executed first and the result will be used by the next transformation in the sequence.

The order in which you perform transformations can make a big difference since you'll want to filter out any collections you don't want to manipulate before you do any complex or intensive operations on them.

Matching Documents

The first stage of the pipeline is matching, and that allows us to filter out documents so that we're only manipulating the documents we care about. The matching expression looks and acts much like the MongoDB find function or a SQL WHERE clause. To find all users that live in Beverly Hills (or more specifically, the 90210 area code), we'll add a match stage to our aggregation pipeline:

```
> db.customers.aggregate([ { $match: { "zip": 90210 }}]);
```

This will return the array of customers that live in the 90210 zip code. Using the match stage in this way is no different from using the find method on a collection. Let's see what kind of insights we can gather by adding some stages to the pipeline.

Grouping Documents

Once we've filtered out the documents we don't want, we can start grouping together the ones that we do into useful subsets. We can also use groups to perform operations across a common field in all documents, such as calculating the sum of a set of transactions and counting documents.

```
> db.customers.aggregate([ { $match: {"zip": "90210"}}, { $group: { _id: null,
    count: { $sum: 1 } } }]);
```

The $group transformation allows us to group documents together and performs transformations or operations across all of those grouped documents. In this case, we're creating a new field in the results called count which adds 1 to a running sum for every document. The _id field is required for grouping and would normally contain fields from each document that we'd like to preserve (ie: phoneNumber). Since we're just looking for the count of every document, we can make it null here.

```
{ "_id" : null, "count" : 24 }
```

Here we saw the use of the $sum arithmetic operator, which sums a field in all of the documents in a collection. We can group our documents together on any fields we'd like and perform other types of computations as well.

Gaining Insights with Sum, Min, Max, and Avg

Transactions are a great place to stretch our proverbial legs when it comes to mathematical operations. A transaction represents a single purchase of one or more products by one customer (many-to-one relationship). An inserted transaction record might look like the following:

Our transaction model looks like this:

```
{
  "id": "1",
  "productId": "1",
  "customerId": "1",
  "amount": 20.00,
  "transactionDate": ISODate("2017-02-23T15:25:56.314Z")
}
```

Let's start by calculating the total amount of sales made for the month of January. We'll start by matching only transactions that occurred between January 1 and January 31.

```
> db.customers.aggregate([ { $match: { transactionDate: {
   $gte: ISODate("2017-01-01T00:00:00.000Z"), $lt: ISODate("2017-02
   01T00:00:00.000Z") } }}]);
```

The $gte and $lt operators are comparators that allow us to limit our dates to specific range. In our case, we want the transactionDate to be greater than or equal to the first day in January and less than the first day in February.

Notice also that we're using Zulu (UTC) time here; you'll want to make sure your transactions are inserted in the proper timezone for this to work, but for this example we'll assume that all timezones are input in Zulu time.

The next stage of the pipeline is summing the transaction amounts and putting that amount in a new field called total:

```
{ $group: {  _id: null,  total: { $sum: "$amount" } }}
```

The final query looks something like this:

```
> db.transactions.aggregate([ {  $match: {

     transactionDate: {$gte: ISODate("2017-01-01T00:00:00.000Z"),
       $lt: ISODate("2017-01-31T23:59:59.000Z") }}},
    {$group: { _id: null, total: { $sum: "$amount" } } }]);
```

The final result is a transaction amount that looks like the following:

```
{ _id: null, total: 20333.00 }
```

Some other helpful monthly metrics we might want are the average price of each transaction, and the minimum and maximum transaction in the month. Let's add those to our group so we can get a single picture of the entire month:

Combined with the $match statement it looks like the following:

```
> db.transactions.aggregate([  {
   $match: { transactionDate: {
      $gte: ISODate("2017-01-01T00:00:00.000Z"),
      $lt: ISODate("2017-01-31T23:59:59.000Z")    }    }  },
    {$group: {  _id: null,  total: {$sum: "$amount" },
     average_transaction_amount: { $avg: "$amount" },
     min_transaction_amount: { $min: "$amount" },
     max_transaction_amount: { $max: "$amount" } } }]);
```

Our final result gives us an interesting picture of what monthly sales looked like in our fictitious cookie shop:

```
{
  _id: null,
  total: 20333.00,
  average_transaction_amount: 8.50,
  min_transaction_amount: 2.99,
  max_transaction_amount: 347.22
}
```

Using these calculations we can see that more than half of our transactions are below $10, and it's likely that our average transaction amount is being skewed by a few outliers (bulk cookie orders).

Now that we know how to use the aggregations pipeline we can start to combine groups, operations, and even other matching parameters to gain deep and detailed insights into any business. It will provide a good launching point for anyone interesting in analyzing data stored in MongoDB.

ASSIGNMENT NO: 04 NoSQL Database

AIM: Map reduce operation with suitable example using MongoDB.

Theory:

Map-reduce is a data processing paradigm for condensing large volumes of data into useful aggregated results. For map-reduce operations, MongoDB provides the mapReduce database command. All map-reduce functions in MongoDB are JavaScript and run within the mongod process. Map-reduce operations take the documents of a single collection as the input and can perform any arbitrary sorting and limiting before beginning the map stage. mapReduce can return the results of a map-reduce operation as a document, or may write the results to collections. The input and the output collections may be sharded. 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.

MapReduce Command: 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:

1. **map** is a javascript function that maps a value with a key and emits a key-value pair
2. **reduce** is a javscript function that reduces or groups all the documents having the same key
3. **out** specifies the location of the map-reduce query result
4. **query** specifies the optional selection criteria for selecting documents
5. **sort** specifies the optional sort criteria
6. **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": "MongoDB is powerful and easy to use DB",
"user_name": "mark",
"status":"active"
}
```

Now, a mapReduce function on posts collection is used 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 result will show a total of documents matched with the query (status:"active"), the map function emitted how many documents with key-value pairs and finally the reduce function grouped mapped documents having the same keys.

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()
```

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

Conclusion: This assignment demonstrates use of Map-Reduce in MongoDB.

********************** QUERIES ***********************

AIM : Map reduce operation with suitable example using MongoDB.

```
> use te_a
switched to db te_a

> show collections
bank
expt5
sales
system.indexes

>
db.bank.insert({"CUSTID":"B1","ACCTYPE":"SB","BRANCH":"PUNE","BALANCE":230
0});
WriteResult({ "nInserted" : 1 })

>
db.bank.insert({"CUSTID":"A1","ACCTYPE":"SB","BRANCH":"PUNE","BALANCE":890
0});
WriteResult({ "nInserted" : 1 })

>
db.bank.insert({"CUSTID":"A1","ACCTYPE":"CU","BRANCH":"PUNE","BALANCE":090
0});
WriteResult({ "nInserted" : 1 })

>
db.bank.insert({"CUSTID":"B1","ACCTYPE":"CU","BRANCH":"PUNE","BALANCE":290
0});
WriteResult({ "nInserted" : 1 })

>db.bank.insert({"CUSTID":"B1","ACCTYPE":"CU","BRANCH":"WAGHOLI","BALANCE"
:2900});
WriteResult({ "nInserted" : 1 })

> db.bank.insert({"CUSTID":"A1","ACCTYPE":"CU","BRANCH":"WAGHOLI","BALANCE
":5600});
WriteResult({ "nInserted" : 1 })

> db.bank.find();
{ "_id" : ObjectId("541e535f313dc145de686dfd"), "CUSTID" : "B1", "ACCTYPE"
: "SB", "BRANCH" : "PUNE", "BALANCE" : 2300 }
{ "_id" : ObjectId("541e536d313dc145de686dfe"), "CUSTID" : "A1", "ACCTYPE"
: "SB", "BRANCH" : "PUNE", "BALANCE" : 8900 }
{ "_id" : ObjectId("541e5379313dc145de686dff"), "CUSTID" : "A1", "ACCTYPE"
: "CU", "BRANCH" : "PUNE", "BALANCE" : 900 }
{ "_id" : ObjectId("541e5386313dc145de686e00"), "CUSTID" : "B1", "ACCTYPE"
: "CU", "BRANCH" : "PUNE", "BALANCE" : 2900 }
{ "_id" : ObjectId("541e5398313dc145de686e01"), "CUSTID" : "B1", "ACCTYPE"
: "CU", "BRANCH" : "WAGHOLI", "BALANCE" : 2900 }
{ "_id" : ObjectId("541e53a3313dc145de686e02"), "CUSTID" : "A1", "ACCTYPE"
: "CU", "BRANCH" : "WAGHOLI", "BALANCE" : 5600 }
```

```
>
db.bank.mapReduce(function(){emit(this.ACCTYPE,1);},function(key,values){r
eturn Array.sum(values)},{out:"TOTAL_BALANCE"});
{
     "result" : "TOTAL_BALANCE",
      "timeMillis" : 5,
      "counts" : {
            "input" : 6,
            "emit" : 6,
            "reduce" : 2,
            "output" : 2
      },
      "ok" : 1,
}

> db.TOTAL_BALANCE.find();

{ "_id" : "CU", "value" : 4 }
{ "_id" : "SB", "value" : 2 }

>
db.bank.mapReduce(function(){emit(this.CUSTID,1);},function(key,values){re
turn Array.sum(values)},{out:"CUST_DET"});
{
      "result" : "CUST_DET",
      "timeMillis" : 4,
      "counts" : {
            "input" : 6,
            "emit" : 6,
            "reduce" : 2,
            "output" : 2
      },
      "ok" : 1,
}

> db.CUST_DET.find();

{ "_id" : "A1", "value" : 3 }
{ "_id" : "B1", "value" : 3 }

>
db.bank.mapReduce(function(){emit(this.CUSTID,1);},function(key,values){re
turn Array.sum(values)},{query:{"BRANCH":"PUNE"},out:"CUST_DET"});

{
      "result" : "CUST_DET",
      "timeMillis" : 5,
      "counts" : {
            "input" : 4,
            "emit" : 4,
            "reduce" : 2,
            "output" : 2
      },
      "ok" : 1,
}
```

```
> db.CUST_DET.find();
{ "_id" : "A1", "value" : 2 }
{ "_id" : "B1", "value" : 2 }


>
db.bank.mapReduce(function(){emit(this.CUSTID,1);},function(key,values){re
turn Array.sum(values)},{query:{"ACCTYPE":"CU"},out:"ACC_DET"});
{
      "result" : "ACC_DET",
      "timeMillis" : 4,
      "counts" : {
            "input" : 4,
            "emit" : 4,
            "reduce" : 2,
            "output" : 2
      },
      "ok" : 1,
}
> db.ACC_DET.find();
{ "_id" : "A1", "value" : 2 }
{ "_id" : "B1", "value" : 2 }
```

ASSIGNMENT NO: 05 NoSQL Database

AIM: Indexing and querying with MongoDB using suitable example.

Theory:

Indexes: 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 requires the mongod 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 index.

The ensureIndex() Method: To create an index you need to use ensureIndex() method of mongodb.

Syntax: Basic syntax of ensureIndex() method is as follows()

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

Here key is the name of filed 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.ensureIndex({“title”:1})

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

>db.mycol.ensureIndex({“title”:1,”description”:-1})

explain() method : This method provides information on the query plan. The query plan is the plan the server uses to find the matches for a query. This information may be useful when optimizing a query. The explain() method returns a document that describes the process used to return the query results. The explain() method has the following form:

db.collection.find().explain()

Querying in MongoDB:

The find() Method : To query data from MongoDB collection, use MongoDB's find() method.

Syntax: Basic syntax of find() method is as follows

```
>db.COLLECTION_NAME.find()
```

find() method will display all the documents in a non structured way.

The pretty() Method: To display the results in a formatted way, you can use pretty() method.

Syntax:

```
>db.mycol.find().pretty()
```

Apart from find() method there is **findOne()** method, that reruns only one document.

AND in MongoDB:

Syntax: In the find() method if you pass multiple keys by separating them by ',' then MongoDB treats it AND condition. Basic syntax of AND is shown below:

```
>db.mycol.find({key1:value1, key2:value2}).pretty()
```

OR in MongoDB:

Syntax: To query documents based on the OR condition, you need to use $or keyword. Basic syntax of OR is shown below:

```
 >db.mycol.find({$or: [{key1: value1}, {key2:value2}]}).pretty()
```

Querying Arrays: Querying for elements of an array is simple. An array can mostly be treated as though each element is the value of the overall key. For example, if the array is a list of fruits,like this:

```
> db.food.insert({"fruit" : ["apple", "banana", "peach"]})
```

the following query:

```
> db.food.find({"fruit" : "banana"})
```

will successfully match the document. We can query for it in much the same way as though we had a document that looked like the (illegal) document: {“fruit” : “apple”, “fruit” : “banana”, “fruit” : “peach”}.

$all: If you need to match arrays by more than one element, you can use “$all”. This allows you to match a list of elements. For example, suppose we created a collection with three elements:

```
> db.food.insert({"_id" : 1, "fruit" : ["apple", "banana", "peach"]})

> db.food.insert({"_id" : 2, "fruit" : ["apple", "kumquat", "orange"]})

> db.food.insert({"_id" : 3, "fruit" : ["cherry", "banana", "apple"]})
```

Then we can find all documents with both “apple” and “banana” elements by querying with **“$all”:**

```
> db.food.find({fruit : {$all : ["apple", "banana"]}})

{"_id" : 1, "fruit" : ["apple", "banana", "peach"]}

{"_id" : 3, "fruit" : ["cherry", "banana", "apple"]}
```

Order does not matter. Notice “banana” comes before “apple” in the second result. Using a one-element array with “$all” is equivalent to not using “$all”. For instance, {fruit : {$all : [‘apple’]} will match the same documents as {fruit : ‘apple’}. You can also query by exact match using the entire array. However, exact match will not match a document if any elements are missing or superfluous. For example, this will match the first document shown previously:

```
> db.food.find({"fruit" : ["apple", "banana", "peach"]})
```

But this will not:

```
> db.food.find({"fruit" : ["apple", "banana"]})
```

Querying on Embedded Documents: There are two ways of querying for an embedded document: querying for the whole document or querying for its individual key/value pairs. Querying for an entire embedded document works identically to a normal query. For example, if we have a document that looks like this:

```
{
“name” : {
“first” : “Joe”,
“last” : “Schmoe”
},
“age” : 45
}
```

we can query for someone named Joe Schmoe with the following:

```
> db.people.find({“name” : {“first” : “Joe”, “last” : “Schmoe”}})
```

However, if Joe decides to add a middle name key, suddenly this query won’t work anymore; it doesn’t match the entire embedded document! This type of query is also order-sensitive; {“last” : “Schmoe”, “first” : “Joe”} would not be a match.

If possible, it’s usually a good idea to query for just a specific key or keys of an embedded document. Then, if your schema changes, all of your queries won’t suddenly break because they’re no longer exact matches. You can query for embedded keys using dotnotation:

```
> db.people.find({“name.first” : “Joe”, “name.last” : “Schmoe”})
```

Now, if Joe adds more keys, this query will still match his first and last names. This dot-notation is the main difference between query documents and other document types. Query documents can contain dots, which mean “reach into an embedded document.” Dot-notation is also the reason that documents to be inserted cannot contain the . character. Oftentimes people run into this limitation when trying to save URLs as keys. One way to get around it is to always perform a global replace before inserting or after retrieving, substituting a character that isn’t legal in URLs for the dot (.) character. Embedded document matches can get a little tricky as the document structure gets more complicated. For example, suppose we are storing blog posts and we want to find comments by Joe that were scored at least a five. We could model the post as follows:

```
> db.blog.find()
{
"content" : "...",
"comments" : [
{
"author" : "joe",
"score" : 3,
"comment" : "nice post"
},
{
"author" : "mary",
"score" : 6,
"comment" : "terrible post"
}
]
}
```

Now, we can't query using

>db.blog.find({"comments" : {"author" : "joe", "score" : {"$gte" : 5}}}).

Embedded document matches have to match the whole document, and this doesn't match the "comment" key. It also wouldn't work to do db.blog.find({"comments.author" : "joe", "comments.score" : {"$gte" : 5}}), because the author criteria could match a different comment than the score criteria. That is, it would return the document shown earlier; it would match "author" : "joe" in the first comment and "score" : 6 in the second comment.

To correctly group criteria without needing to specify every key, use "$elemMatch". This vaguely named conditional allows you to partially specify criteria to match a single embedded document in an array. The correct query looks like this:

> db.blog.find({"comments" : {"$elemMatch" : {"author" : "joe", "score" : {"$gte" : 5}}}})

Conclusion:

This assignment explains details about use of indexing and querying in MongoDB.

******************************QUERIES******************

AIM : Indexing and querying with MongoDB using suitable example.

```
> use te_a
switched to db te_a

> show collections
expt5
system.indexes

> db.expt5.find().pretty();
{
     "_id" : ObjectId("541cff058a5439b34ba0b41d"),
      "RNO" : 3,
      "NAME" : {
            "FIRST" : "AKASH",
            "LNAME" : "KOTHARI"
      },
      "ADDRESS" : {
            "CITY" : "PUNE",
            "DISTRICT" : "PUNE",
            "PIN" : 444605
      },
      "CONTACTNO" : 8888888888,
      "EMAIL" : "jfsdkfj@dkfj.com"
}
{
      "_id" : ObjectId("541e3d3d6a3acb31eb494c35"),
      "RNO" : 2,
      "NAME" : {
            "FIRST" : "ISHAAN",
            "LNAME" : "KOTHARI"
      },
      "ADDRESS" : {
            "CITY" : "AMRAVATI",
            "DISTRICT" : "AMRAVATI",
            "PIN" : 444605
      },
      "CONTACTNO" : 789798797
}
{
      "_id" : ObjectId("541e4391313dc145de686df9"),
      "RNO" : 1,
      "NAME" : {
            "FIRST" : "SANDESH",
            "LNAME" : "GANGWAL"
      },
      "ADDRESS" : {
            "CITY" : "PUNE",
            "DISTRICT" : "PUNE",
            "PIN" : 345678
      },
      "CONTACTNO" : [
              1234567,
              456788076
       ]
}

> db.expt5.ensureIndex({"RNO":1},{"UNIQUE":true});
{
       "createdCollectionAutomatically" : false,
       "numIndexesBefore" : 1,
       "numIndexesAfter" : 2,
       "ok" : 1
}

> db.expt5.find().explain();
{
       "cursor" : "BasicCursor",
       "isMultiKey" : false,
       "n" : 3,
       "nscannedObjects" : 3,
       "nscanned" : 3,
       "nscannedObjectsAllPlans" : 3,
       "nscannedAllPlans" : 3,
       "scanAndOrder" : false,
       "indexOnly" : false,
       "nYields" : 0,
       "nChunkSkips" : 0,
       "millis" : 0,
       "server" : "localhost.localdomain:27017",
       "filterSet" : false
}

> db.expt5.insert({"RNO" : 1, "NAME" : { "FIRST" : "SANDESH", "LNAME" :
"GANGWAL" },
"ADDRESS" : { "CITY" : "PUNE", "DISTRICT" : "PUNE", "PIN" : 345678 },
"CONTACTNO" :[1234567,456788076] });
WriteResult({
       "nInserted" : 0,
       "writeError" : {
              "code" : 11000,
              "errmsg" : "insertDocument :: caused by :: 11000 E11000
duplicate key error index: te_a.expt5.$RNO_1  dup key: { : 1.0 }"
       }
})

> db.expt5.find().sort({"NAME.FIRST":1});
{ "_id" : ObjectId("541cff058a5439b34ba0b41d"), "RNO" : 3, "NAME" : {
"FIRST" : "AKASH", "LNAME" : "KOTHARI" }, "ADDRESS" : { "CITY" : "PUNE",
"DISTRICT" : "PUNE", "PIN" : 444605 }, "CONTACTNO" : 8888888888, "EMAIL" :
"jfsdkfj@dkfj.com" }
{ "_id" : ObjectId("541e3d3d6a3acb31eb494c35"), "RNO" : 2, "NAME" : {
"FIRST" : "ISHAAN", "LNAME" : "KOTHARI" }, "ADDRESS" : { "CITY" :
"AMRAVATI", "DISTRICT" : "AMRAVATI", "PIN" : 444605 }, "CONTACTNO" :
789798797 }
{ "_id" : ObjectId("541e4391313dc145de686df9"), "RNO" : 1, "NAME" : {
"FIRST" : "SANDESH", "LNAME" : "GANGWAL" }, "ADDRESS" : { "CITY" : "PUNE",
"DISTRICT" : "PUNE", "PIN" : 345678 }, "CONTACTNO" : [ 1234567, 456788076
] }

> db.expt5.find({"RNO":3});
{ "_id" : ObjectId("541cff058a5439b34ba0b41d"), "RNO" : 3, "NAME" : {
"FIRST" : "AKASH", "LNAME" : "KOTHARI" }, "ADDRESS" : { "CITY" : "PUNE",
"DISTRICT" : "PUNE", "PIN" : 444605 }, "CONTACTNO" : 8888888888, "EMAIL" :
"jfsdkfj@dkfj.com" }

> db.expt5.find({"RNO":3,"PIN":3434434});


> db.expt5.find({"RNO":3,"NAME.FIRST":"AKASH"});
{ "_id" : ObjectId("541cff058a5439b34ba0b41d"), "RNO" : 3, "NAME" : {
"FIRST" : "AKASH", "LNAME" : "KOTHARI" }, "ADDRESS" : { "CITY" : "PUNE",
"DISTRICT" : "PUNE", "PIN" : 444605 }, "CONTACTNO" : 8888888888, "EMAIL" :
"jfsdkfj@dkfj.com" }

> db.expt5.find({$or:[{"RNO":3},{"ADDRESS.CITY":"PUNE"}]});
{ "_id" : ObjectId("541cff058a5439b34ba0b41d"), "RNO" : 3, "NAME" : {
"FIRST" : "AKASH", "LNAME" : "KOTHARI" }, "ADDRESS" : { "CITY" : "PUNE",
"DISTRICT" : "PUNE", "PIN" : 444605 }, "CONTACTNO" : 8888888888, "EMAIL" :
"jfsdkfj@dkfj.com" }
{ "_id" : ObjectId("541e4391313dc145de686df9"), "RNO" : 1, "NAME" : {
"FIRST" : "SANDESH", "LNAME" : "GANGWAL" }, "ADDRESS" : { "CITY" : "PUNE",
"DISTRICT" : "PUNE", "PIN" : 345678 }, "CONTACTNO" : [ 1234567, 456788076
] }

 >db.expt5.find({"RNO":{$gt:1}});
{ "_id" : ObjectId("541e3d3d6a3acb31eb494c35"), "RNO" : 2, "NAME" : {
"FIRST" : "ISHAAN", "LNAME" : "KOTHARI" }, "ADDRESS" : { "CITY" :
"AMRAVATI", "DISTRICT" : "AMRAVATI", "PIN" : 444605 }, "CONTACTNO" :
789798797 }
{ "_id" : ObjectId("541cff058a5439b34ba0b41d"), "RNO" : 3, "NAME" : {
"FIRST" : "AKASH", "LNAME" : "KOTHARI" }, "ADDRESS" : { "CITY" : "PUNE",
"DISTRICT" : "PUNE", "PIN" : 444605 }, "CONTACTNO" : 8888888888, "EMAIL" :
"jfsdkfj@dkfj.com" }

> db.expt5.find().sort({"NAME.FIRST":1});
{ "_id" : ObjectId("541cff058a5439b34ba0b41d"), "RNO" : 3, "NAME" : {
"FIRST" : "AKASH", "LNAME" : "KOTHARI" }, "ADDRESS" : { "CITY" : "PUNE",
"DISTRICT" : "PUNE", "PIN" : 444605 }, "CONTACTNO" : 8888888888, "EMAIL" :
"jfsdkfj@dkfj.com" }
{ "_id" : ObjectId("541e3d3d6a3acb31eb494c35"), "RNO" : 2, "NAME" : {
"FIRST" : "ISHAAN", "LNAME" : "KOTHARI" }, "ADDRESS" : { "CITY" :
"AMRAVATI", "DISTRICT" : "AMRAVATI", "PIN" : 444605 }, "CONTACTNO" :
789798797 }
{ "_id" : ObjectId("541e4391313dc145de686df9"), "RNO" : 1, "NAME" : {
"FIRST" : "SANDESH", "LNAME" : "GANGWAL" }, "ADDRESS" : { "CITY" : "PUNE",
"DISTRICT" : "PUNE", "PIN" : 345678 }, "CONTACTNO" : [ 1234567, 456788076
] }

> db.expt5.find().sort({"ADDRESS.CITY":-1})
{ "_id" : ObjectId("541e4391313dc145de686df9"), "RNO" : 1, "NAME" : {
"FIRST" : "SANDESH", "LNAME" : "GANGWAL" }, "ADDRESS" : { "CITY" : "PUNE",
"DISTRICT" : "PUNE", "PIN" : 345678 }, "CONTACTNO" : [ 1234567, 456788076
] }
{ "_id" : ObjectId("541cff058a5439b34ba0b41d"), "RNO" : 3, "NAME" : {
"FIRST" : "AKASH", "LNAME" : "KOTHARI" }, "ADDRESS" : { "CITY" : "PUNE",
"DISTRICT" : "PUNE", "PIN" : 444605 }, "CONTACTNO" : 8888888888, "EMAIL" :
"jfsdkfj@dkfj.com" }
{ "_id" : ObjectId("541e3d3d6a3acb31eb494c35"), "RNO" : 2, "NAME" : {
"FIRST" : "ISHAAN", "LNAME" : "KOTHARI" }, "ADDRESS" : { "CITY" :
"AMRAVATI", "DISTRICT" : "AMRAVATI", "PIN" : 444605 }, "CONTACTNO" :
789798797 }

> db.expt5.find().sort({"ADDRESS.CITY":-1,"NAME.FIRST":1});
{ "_id" : ObjectId("541cff058a5439b34ba0b41d"), "RNO" : 3, "NAME" : {
"FIRST" : "AKASH", "LNAME" : "KOTHARI" }, "ADDRESS" : { "CITY" : "PUNE",
"DISTRICT" : "PUNE", "PIN" : 444605 }, "CONTACTNO" : 8888888888, "EMAIL" :
"jfsdkfj@dkfj.com" }
{ "_id" : ObjectId("541e4391313dc145de686df9"), "RNO" : 1, "NAME" : {
"FIRST" : "SANDESH", "LNAME" : "GANGWAL" }, "ADDRESS" : { "CITY" : "PUNE",
"DISTRICT" : "PUNE", "PIN" : 345678 }, "CONTACTNO" : [ 1234567, 456788076
] }
{ "_id" : ObjectId("541e3d3d6a3acb31eb494c35"), "RNO" : 2, "NAME" : {
"FIRST" : "ISHAAN", "LNAME" : "KOTHARI" }, "ADDRESS" : { "CITY" :
"AMRAVATI", "DISTRICT" : "AMRAVATI", "PIN" : 444605 }, "CONTACTNO" :
789798797 }

> db.expt5.find().sort({"FNAME":1});
{ "_id" : ObjectId("541e3d3d6a3acb31eb494c35"), "RNO" : 2, "NAME" : {
"FIRST" : "ISHAAN", "LNAME" : "KOTHARI" }, "ADDRESS" : { "CITY" :
"AMRAVATI", "DISTRICT" : "AMRAVATI", "PIN" : 444605 }, "CONTACTNO" :
789798797 }
{ "_id" : ObjectId("541e4391313dc145de686df9"), "RNO" : 1, "NAME" : {
"FIRST" : "SANDESH", "LNAME" : "GANGWAL" }, "ADDRESS" : { "CITY" : "PUNE",
"DISTRICT" : "PUNE", "PIN" : 345678 }, "CONTACTNO" : [ 1234567, 456788076
] }
{ "_id" : ObjectId("541cff058a5439b34ba0b41d"), "RNO" : 3, "NAME" : {
"FIRST" : "AKASH", "LNAME" : "KOTHARI" }, "ADDRESS" : { "CITY" : "PUNE",
"DISTRICT" : "PUNE", "PIN" : 444605 }, "CONTACTNO" : 8888888888, "EMAIL" :
"jfsdkfj@dkfj.com" }

>  db.expt5.find().skip(2);
{ "_id" : ObjectId("541e4391313dc145de686df9"), "RNO" : 1, "NAME" : {
"FIRST" : "SANDESH", "LNAME" : "GANGWAL" }, "ADDRESS" : { "CITY" : "PUNE",
"DISTRICT" : "PUNE", "PIN" : 345678 }, "CONTACTNO" : [ 1234567, 456788076
] }


>  db.expt5.find().limit(2);
{ "_id" : ObjectId("541cff058a5439b34ba0b41d"), "RNO" : 3, "NAME" : {
"FIRST" : "AKASH", "LNAME" : "KOTHARI" }, "ADDRESS" : { "CITY" : "PUNE",
"DISTRICT" : "PUNE", "PIN" : 444605 }, "CONTACTNO" : 8888888888, "EMAIL" :
"jfsdkfj@dkfj.com" }
{ "_id" : ObjectId("541e3d3d6a3acb31eb494c35"), "RNO" : 2, "NAME" : {
"FIRST" : "ISHAAN", "LNAME" : "KOTHARI" }, "ADDRESS" : { "CITY" :
"AMRAVATI", "DISTRICT" : "AMRAVATI", "PIN" : 444605 }, "CONTACTNO" :
789798797 }
```