Assignment No.

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 multiple documents, 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)

Objectives: To understand all basic operations and administration commands of MongoDB

Theory:-

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:

>usemydb

switched to dbmydb

To check your currently selected database use the command db

>db

mydb

If you want to check your databases list, then use the command show dbs.

>show dbs

local0.78125GB

test0.23012GB

Your created database (mydb) is not present in list. To display database you need to insert atleast one document into it.

>db.movie.insert({"name":"ABC"})

>show dbs

local0.78125GB

mydb0.23012GB

test0.23012GB

In mongodb default database is test. If you didn't create any database then collections will be stored in test database.

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
local0.78125GB
mydb0.23012GB
test0.23012GB
>
If you want to delete new database <mydb>, then dropDatabase() command would be as follows:
>usemydb
switched to dbmydb
>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

Parameter	Type	Description
Name	String	Name of the collection to be created
Options	Document	(Optional) Specify options about memory size and indexing

Options parameter is optional, so you need to specify only name of the collection. Following is the list of options you can use:

Field	Type	Description
capped	Boolean	(Optional) If true, enables a capped collection. Capped collection is a collection fixed size collecction that automatically overwrites its oldest entries when it reaches its maximum size. If you specify true, you need to specify size parameter also.
autoIndexID	Boolean	(Optional) If true, automatically create index on _id field.s Default value is false.
size	number	(Optional) Specifies a maximum size in bytes for a capped collection. If If capped is true, then you need to specify this field also.
max	number	(Optional) Specifies the maximum number of documents allowed in the

		capped collection.

While inserting the document, MongoDB first checks size field of capped collection, then it checks max field.
Examples:
Basic syntax of createCollection() method without options is as follows
>use test
switched to db test
>db.createCollection("mycollection")
{"ok":1}
>

The drop() Method
MongoDB'sdb.collection.drop() is used to drop a collection from the database.
Syntax:
Basic syntax of drop() command is as follows
db.COLLECTION_NAME.drop()

DATA TYPES OF MONGODB:-
MongoDB supports many datatypes whose list is given below:
String : This is most commonly used datatype to store the data. String in mongodb must be UTF-8 valid.
Integer : This type is used to store a numerical value. Integer can be 32 bit or 64 bit depending upon your server.
Boolean : This type is used to store a boolean (true/ false) value.
Double : This type is used to store floating point values.
Min/ Max keys : This type is used to compare a value against the lowest and highest BSON elements.
Arrays : This type is used to store arrays or list or multiple values into one key.
Timestamp :ctimestamp. This can be handy for recording when a document has been modified or added.
Object : This datatype is used for embedded documents.
Null : This type is used to store a Null value.
Symbol : This datatype is used identically to a string however, it's generally reserved for languages that use a specific symbol type.
Date : This datatype is used to store the current date or time in UNIX time format. You can specify your own date time by creating object of Date and passing day, month, year into it.
Object ID : This datatype is used to store the document's ID.

Binary data : This datatype is used to store binay data.
Code : This datatype is used to store javascript code into document.
Regular expression : This datatype is used to store regular expression

The insert() Method
To insert data into MongoDB collection, you need to use MongoDB'sinsert() 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',
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.
In the inserted document if we don't specify the _id parameter, then MongoDB assigns an unique ObjectId for this document.
_id is 12 bytes hexadecimal number unique for every document in a collection. 12 bytes are divided as follows:
_id:ObjectId(4 bytes timestamp,3 bytes machine id,2 bytes process id,3 bytes incrementer)

MongoDB – Bulk.insert() Method

MongoDB, the Bulk.insert() method is used to perform insert operations in bulk.
The Bulk.insert() method is used to insert multiple documents in one go.

```
var bulk = db.students.initializeUnorderedBulkOp();
bulk.insert( { first_name: "Sachin", last_name: "Tendulkar" } );
bulk.insert( { first_name: "Virender", last_name: "Sehwag" } );
bulk.insert( { first_name: "Shikhar", last_name: "Dhawan" } );
bulk.insert( { first_name: "Mohammed", last_name: "Shami" } );
bulk.insert( { first_name: "Shreyas", last_name: "Iyer" } );
bulk.execute();
```

Unordered Insertion of documents:
db.students.find().sort({'_id':-1}).limit(5).pretty()

Schema Validation in MongoDB

- Schema validation in MongoDB provides a structured approach to define and enforce rules for document structures within collections.
- By specifying validation criteria such as data types, required fields, and custom expressions using JSON Schema syntax, MongoDB ensures data integrity and consistency.
- Mongodb schema validation
- Schema validation in MongoDB is a feature that allows us to set the structure for the data in the documents of a collection.
- We follow some set of rules, and validation rules, which ensure that the data we insert or update follows a specific predefined schema and ensures the data must have only specific datatypes, required fields, and validation expressions mentioned in the predefined schema.
- When we create a collection for the first time and we want it to meet specific criteria then we can define the collection with the schema validation rules.
- These validation rules can include specifying the required fields we want, and the datatype for those fields, and also allow the user's custom expressions. We use the command $jsonSchema for specifying the rules.

When to use Schema Validation

- Schema Validation is like setting rules for how your document must look in our database.
- When we are experimenting on a new application in which we think that the incoming data might change the application's fields and we are unsure about the structure we might not want to use schema validation.
- Step 1: We create a collection named of 'students' using the createCollection() command.
- Step 2: With the '$jsonShema' command inside the validator we specify the schema validation rules.
- Here with the required property we give a list of fields that every document must have when inserted into the collection.

- Step 3: Give all the fields and their datatypes inside the properties.

```
db.createCollection("Students", {
    validator: {
        $jsonSchema: {
            bsonType: "object",
            required: ["name", "id"],
            properties: {
                name: {
                    bsonType: "string",
                    description: "Name must be a string."
                },
                id: {
                   bsonType: "int",
                    description: "id must be an integer."
                },

            }
        }
    }
});
show collections;
db.Students.insert({name:"s",id:1})
db.Students.find()
```

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)
Example
Consider the mycolcollectioin has following data.
{"_id":ObjectId(5983548781331adf45ec5),"title":"MongoDB Overview"}
{"_id":ObjectId(5983548781331adf45ec6),"title":"NoSQL Overview"}
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'}})

```
>db.mycol.find()
{"_id":ObjectId(5983548781331adf45ec5),"title":"New MongoDB Tutorial"}
{"_id":ObjectId(5983548781331adf45ec6),"title":"NoSQL Overview"}
>
```

The remove() Method

MongoDB'sremove() method is used to remove document from the collection. remove() method accepts two parameters. One is deletion criteria and second is justOne flag
deletion criteria : (Optional) deletion criteria according to documents will be removed.
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)
```

Find Method(Querying):-

The find method is used to perform queries in MongoDB. Querying returns a subset of documents in a collection, from no documents at all to the entire collection. Which documents get returned is determined by the first argument to find, which is a document specifying the query to be performed.
An empty query document (i.e., {}) matches everything in the collection. If find isn't given a query document, it defaults to {}. For example, the following:

```
>db.c.find()
```

returns everything in the collection c.
Querying for a simple type is as easy as specifying the value that you are looking for. For example, to find all documents where the value for "age" is 27, we can add that key/value pair to the query document:

```
>db.users.find({"age" : 27})
```

If we have a string we want to match, such as a "username" key with the value "abc",we use that key/value pair instead:

```
>db.users.find({"username" : "abc"})
```

Multiple conditions can be strung together by adding more key/value pairs to the query document, which gets interpreted as "condition1 AND condition2 AND … AND conditionN." For instance, to get all users who are 27-year-olds with the username "abc" we can query for the following:

```
>db.users.find({"username" : "abc", "age" : 27})
```

Query Criteria

Queries can go beyond the exact matching described in the previous section; they can match more complex criteria, such as ranges, OR-clauses, and negation.

Query Conditionals

"$lt", "$lte", "$gt", and "$gte" are all comparison operators, corresponding to <, <=, >, and >=, respectively. They can be combined to look for a range of values. For example, to look for users who are between the ages of 18 and 30 inclusive, we can do this:

```
>db.users.find({"age" : {"$gte" : 18, "$lte" : 30}})
```

These types of range queries are often useful for dates. For example, to find people who registered before January 1, 2007, we can do this:

```
> start = new Date("01/01/2007")
>db.users.find({"registered" : {"$lt" : start}})
```

An exact match on a date is less useful, because dates are only stored with millisecond precision. Often you want a whole day, week, or month, making a range query necessary. To query for documents where a key's value is not equal to a certain value, you must use another conditional operator, "$ne", which stands for "not equal." If you want to find all users who do not have the username "abc" you can query for them using this:
>db.users.find({"username" : {"$ne" : "abc"}})"$ne" can be used with any type.

OR Queries

There are two ways to do an OR query in MongoDB. "$in" can be used to query for a variety of values for a single key. "$or" is more general; it can be used to query for any of the given values across multiple keys.
If you have more than one possible value to match for a single key, use an array of criteria with "$in". For instance, suppose we were running a train and the winning ticket numbers were 725, 542, and 390. To find all three of these documents, we can construct the following query:

```
>db.train.find({"ticket_no" : {"$in" : [725, 542, 390]}})
```

- **In Operator**

"$in"is very flexible and allows you to specify criteria of different types as well as values. For example, if we are gradually migrating our schema to use usernames instead of user ID numbers, we can query for either by using this:

```
>db.users.find({"user_id" : {"$in" : [12345, "abc"]})
```

This matches documents with a "user_id" equal to 12345, and documents with a "user_id" equal to "abc".If "$in" is given an array with a single value, it behaves the same as directly matching the value. For instance, {ticket_no : {$in : [725]}} matches the same documents as {ticket_no : 725}.
The opposite of "$in" is "$nin", which returns documents that don't match any of the criteria in the array. If we want to return all of the people who didn't win anything in the train, we can query for them with this:

```
>db.train.find({"ticket_no" : {"$nin" : [725, 542, 390]}})
```

This query returns everyone who did not have tickets with those numbers. "$in" gives you an OR query for a single key, but what if we need to find documents where "ticket_no" is

725 or "winner" is true? For this type of query, we'll need to use the "$or" conditional. "$or" takes an array of possible criteria. In the train case, using "$or" would look like this:

```
>db.train.find({"$or" : [{"ticket_no" : 725}, {"winner" : true}]})
```

"$or"can contain other conditionals. If, for example, we want to match any of the three "ticket_no" values or the "winner" key, we can use this:

```
>db.train.find({"$or" : [{"ticket_no" : {"$in" : [725, 542, 390]}}, {"winner" : true}]})
```

With a normal AND-type query, you want to narrow your results down as far as possible in as few arguments as possible. OR-type queries are the opposite: they are most efficient if the first arguments match as many documents as possible.

- **$not operator**

"$not" is a metaconditional: it can be applied on top of any other criteria. As an example, let's consider the modulus operator, "$mod". "$mod" queries for keys whose values, when divided by the first value given, have a remainder of the second value:

```
>db.users.find({"id_num" : {"$mod" : [5, 1]}})
```

The previous query returns users with "id_num"s of 1, 6, 11, 16, and so on. If we want,instead, to return users with "id_num"s of 2, 3, 4, 5, 7, 8, 9, 10, 12, and so on, we can use "$not":

```
>db.users.find({"id_num" : {"$not" : {"$mod" : [5, 1]}}})
```

"$not" can be particularly useful in conjunction with regular expressions to find all documents that don't match a given pattern

- **Limits, Skips, and Sorts**

The most common query options are limiting the number of results returned, skipping a number of results, and sorting. All of these options must be added before a query is sent to the database.
To set a limit, chain the limit function onto your call to find. For example, to only return three results, use this:

```
>db.c.find().limit(3)
```

If there are fewer than three documents matching your query in the collection, only the number of matching documents will be returned; limit sets an upper limit, not a lower limit.skipworks similarly to limit:

```
>db.c.find().skip(3)
```

This will skip the first three matching documents and return the rest of the matches. Ifthere are less than three documents in your collection, it will not return any documents.sorttakes an object: a set of key/value pairs where the keys are key names and thevalues are the sort directions. Sort direction can be 1 (ascending) or -1 (descending). IfCursors | 57multiple keys are given, the results will be sorted in that order. For instance, to sort theresults by "username" ascending and "age" descending, we do the following:

>db.c.find().sort({username : 1, age : -1})

These three methods can be combined. This is often handy for pagination. For example,suppose that you are running an online store and someone searches for mp3. If youwant 50 results per page sorted by price from high to low, you can do the following:

>db.stock.find({"desc" : "mp3"}).limit(50).sort({"price" : -1})

If they click Next Page to see more results, you can simply add a skip to the query,which will skip over the first 50 matches (which the user already saw on page 1):

>db.stock.find({"desc" : "mp3"}).limit(50).skip(50).sort({"price" : -1})

However, large skips are not very performant, so there are suggestions on avoidingthem in a moment.

- **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"]})
and neither will this:
>db.food.find({"fruit" : ["banana", "apple", "peach"]})

$size

A useful conditional for querying arrays is "$size", which allows you to query for arrays of a given size. Here's an example:

```
>db.food.find({"fruit" : {"$size" : 3}})
```

One common query is to get a range of sizes. "$size" cannot be combined with another $ conditional (in this example, "$gt"), but this query can be accomplished by adding a "size" key to the document. Then, every time you add an element to the array, increment the value of "size". If the original update looked like this:
>db.food.update({"$push" : {"fruit" : "strawberry"}})
it can simply be changed to this:
>db.food.update({"$push" : {"fruit" : "strawberry"}, "$inc" : {"size" : 1}})

Incrementing is extremely fast, so any performance penalty is negligible. Storing documents like this allows you to do queries such as this:

```
>db.food.find({"size" : {"$gt" : 3}})
```

Unfortunately, this technique doesn't work as well with the "$addToSet" operator. The $slice operator As mentioned earlier in this chapter, the optional second argument to find specifies the keys to be returned. The special "$slice" operator can be used to return a subset of elements for an array key.
For example, suppose we had a blog post document and we wanted to return the first 10 comments:

```
>db.blog.posts.findOne(criteria, {"comments" : {"$slice" : 10}})
```

Alternatively, if we wanted the last 10 comments, we could use -10:

```
>db.blog.posts.findOne(criteria, {"comments" : {"$slice" : -10}})
```

"$slice" can also return pages in the middle of the results by taking an offset and the number of elements to return:

```
>db.blog.posts.findOne(criteria, {"comments" : {"$slice" : [23, 10]}})
```

This would skip the first 23 elements and return the 24th through 34th. If there are fewer than 34 elements in the array, it will return as many as possible.

- **Pattern Matching Using $regex Operator**

- This operator provides regular expression capabilities for pattern-matching stings in the queries.
- In other words, this operator is used to search for the given string in the specified collection.
- t is helpful when we don't know the exact field value that we are looking in the document.
- For example, a collection containing 3 documents i.e.,

- **Important Points:**
  - You are not allowed to use $regex operator inside $in operator.
  - If you want to add a regular expression inside a comma-separated list of a query condition, then you have to use the $regex operator.
  - If you want to use x and s options then you have to use $regex operator expression with $options.
  - Starting from the latest version of MongoDB(i.e., 4.0.7) you are allowed to use $not operator with $regex operator expression.
  - For case-sensitive regular expression queries, if the index of the specified field is available, then MongoDB matches the regular expression to the values in the index. It is the easiest way to match rather than scanning all the collections For case-insensitive regular expression queries, they do not utilize index effectively.
  - If you want to use Perl compatible regular expressions support regular expressions that are not supported in JavaScript, then you must use $regex operator.
- **Syntax**
  - { <field>: { $regex: /pattern/, $options: '<options>' } }
  - { <field>: { $regex: 'pattern', $options: '<options>' } }
  - **$options:**
  - In MongoDB, the following <options> are available for use with regular expression:
  - i: To match both **lower case and upper case** pattern in the string.
  - m: To include ^ and $ in the pattern in the match i.e. to specifically search for ^ and $ inside the string. Without this option, these anchors match at the beginning or end of the string.
  - x: To ignore all **white space** characters in the $regex pattern.
  - s: To allow the dot character **"." to match all characters including newline characters.**
- **Using $where Operator**
  - The $where operator allows you to use JavaScript expressions to filter documents.
  - This approach can be less efficient than other methods and should be used with caution, especially on large datasets.

- The $where operator allows you to use JavaScript expressions for more complex queries.
- Note that using $where can impact performance and should be used with caution.
- Example:

```
db.collection.find({  $where: function()
{    return this.name.includes("abc");  }})
```

```
db.collection.find({ $where: function() { return this.name.indexOf("abc") !== -1; } })
```

The line return this.name.indexOf("abc") !== -1; is a JavaScript expression used within MongoDB's $where operator to filter documents based on whether the name field contains the substring "abc". Here's a breakdown of what this line does:

Breakdown of the Expression

1. **this.name**:
   - Refers to the name field of the current document being evaluated.
2. **.indexOf("abc")**:
   - The indexOf method is a built-in JavaScript string method that searches for the specified substring ("abc" in this case) within the string it is called on (this.name).
   - It returns the index (i.e., the position) of the first occurrence of the specified substring.
   - If the substring is not found, it returns -1.
3. **!== -1**:
   - This is a strict inequality comparison operator.
   - It checks if the result of indexOf("abc") is not equal to -1.
   - If the substring "abc" is found within this.name, indexOf returns a value other than -1, and the expression evaluates to true.
   - If the substring "abc" is not found, indexOf returns -1, and the expression evaluates to false.

Conclusion: Successfully we have executed Mongodb CRUD operation.

Assignment No. 2(A)

Problem statement: indexing with suitable example using MongoDB.

Objectives: To understand the basic indexing of MongoDB.

INDEXING:-

Indexes support the efficient execution of queries in MongoDB. Without indexes, MongoDB must scan every document in a collection to select those documents that match the query statement. These *collection scans* are inefficient because they require mongod to process a larger volume of data than an index for each operation.
Indexes are special data structures that store a small portion of the collection's 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.
Fundamentally, indexes in MongoDB are similar to indexes in other database systems. MongoDB defines indexes at the collection level and supports indexes on any field or sub-field of the documents in a MongoDB collection.
If an appropriate index exists for a query, MongoDB can use the index to limit the number of documents it must inspect. In some cases, MongoDB can use the data from the index to determine which documents match a query. The following diagram illustrates a query that selects documents using an index.

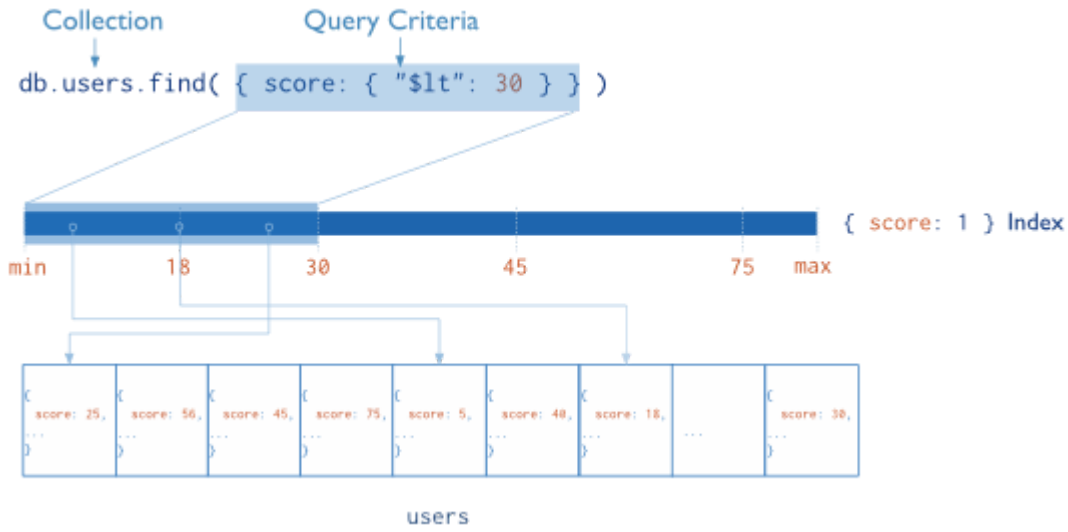

Diagram of a query selecting documents using an index. MongoDB narrows the query by scanning the range of documents with values of score less than 30.
Index Types
MongoDB provides a number of different index types to support specific types of data and queries.
Default _id
All MongoDB collections have an index on the _id field that exists by default. If applications do not specify a value for _id the driver or the mongod will create an_id field with an *ObjectId*value.
The _id index is *unique*, and prevents clients from inserting two documents with the same value for the _id field.

Single Field
In addition to the MongoDB-defined _id index, MongoDB supports user-defined indexes on a *single field of a document*. Consider the following illustration of a single-field index:

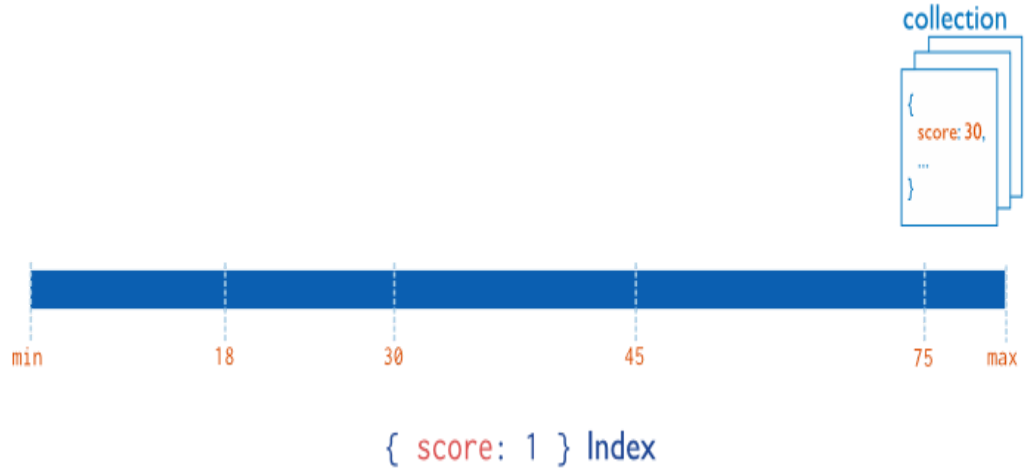

Diagram of an index on the score field (ascending).

Compound Index

MongoDB *also* supports user-defined indexes on multiple fields. These *compound indexes* behave like single-field indexes; *however*, the query can select documents based on additional fields. The order of fields listed in a compound index has significance. For instance, if a compound index consists of {userid:1,score:-1}, the index sorts first by userid and then, within each userid value, sort by score. Consider the following illustration of this compound index:

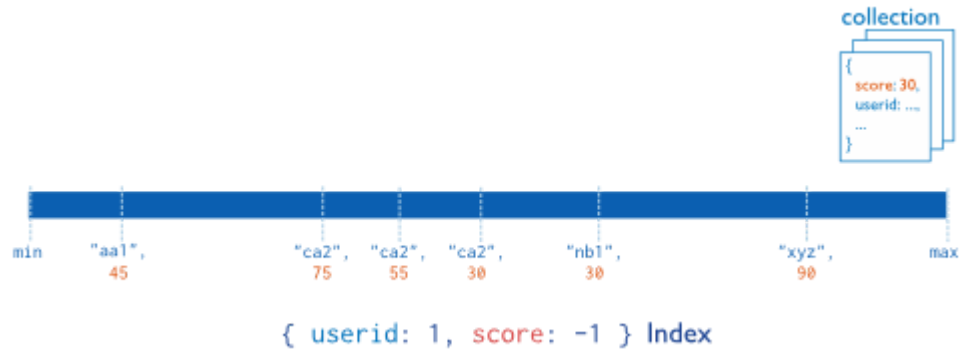

Diagram of a compound index on the userid field (ascending) and the score field (descending). The index sorts first by the userid field and then by the score field.

Multikey Index

MongoDB uses *multikey indexes*to index the content stored in arrays. If you index a field that holds an array value, MongoDB creates separate index entries for *every* element of the array. These *multikey indexes* allow queries to select documents that contain arrays by matching on element or elements of the arrays. MongoDB automatically determines whether to create a multikey index if the indexed field contains an array value; you do not need to explicitly specify the multikey type.
Consider the following illustration of a multikey index:

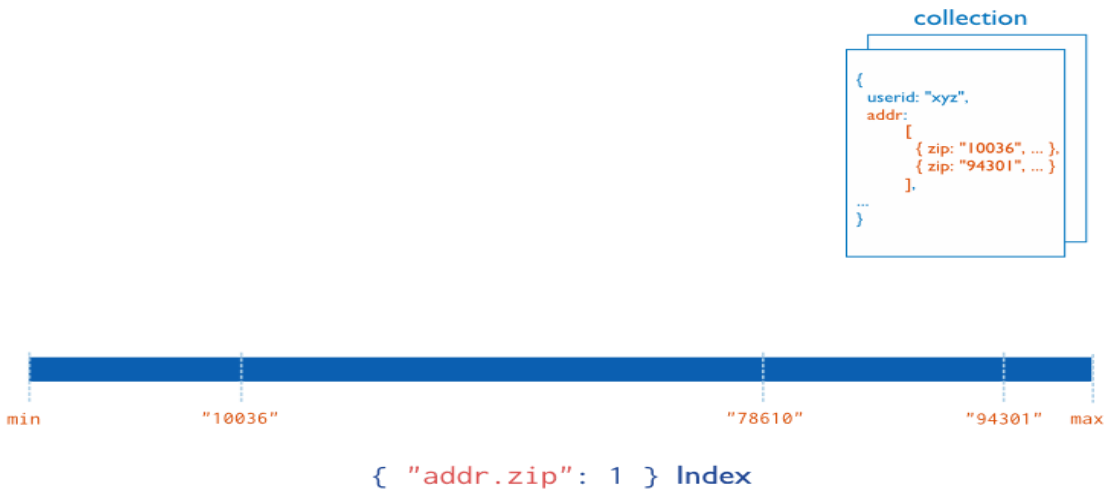

Diagram of a multikey index on the addr.zip field. The addr field contains an array of address documents. The address documents contain the zip field.

Geospatial Index

To support efficient queries of geospatial coordinate data, MongoDB provides two special indexes: *2d indexes* that uses planar geometry when returning results and *2sphere indexes* that use spherical geometry to return results.

Text Indexes

MongoDB provides a text index type that supports searching for string content in a collection. These text indexes do not store language-specific *stop* words (e.g. “the”, “a”, “or”) and *stem* the words in a collection to only store root words.

Hashed Indexes

To support *hash based sharding*, MongoDB provides a *hashed index* type, which indexes the hash of the value of a field. These indexes have a more random distribution of values along their range, but *only* support equality matches and cannot support range-based queries.

Index Properties

Unique Indexes

The *unique* property for an index causes MongoDB to reject duplicate values for the indexed field. To create a *unique index*on a field that already has duplicate values,. Other than the unique constraint, unique indexes are functionally interchangeable with other MongoDB indexes.

Sparse Indexes

The *sparse*property of an index ensures that the index only contain entries for documents that have the indexed field. The index skips documents that *do not* have the indexed field.

You can combine the sparse index option with the unique index option to reject documents that have duplicate values for a field but ignore documents that do not have the indexed key.

Create an Index

Indexes allow MongoDB to process and fulfill queries quickly by creating small and efficient representations of the documents in a *collection*. Users can create indexes for any collection on any field in a *document*. By default, MongoDB creates an index on the _id field of every collection.

Create an Index on a Single Field

To create an index, use ensureIndex() or a similar method from your driver. The ensureIndex() method only creates an index if an index of the same specification does not already exist.

For example, the following operation creates an index on the userid field of the records collection:

db.records.ensureIndex({userid:1})
The value of the field in the index specification describes the kind of index for that field. For example, a value of 1 specifies an index that orders items in ascending order. A value of -1 specifies an index that orders items in descending order.
The created index will support queries that select on the field userid, such as the following:
db.records.find({userid:2})
db.records.find({userid:{$gt:10}})
But the created index does not support the following query on the profile_url field:
db.records.find({profile_url:2})
For queries that cannot use an index, MongoDB must scan all documents in a collection for documents that match the query.

Create a Compound Index

Indexes allow MongoDB to process and fulfill queries quickly by creating small and efficient representations of the documents in a *collection*. MongoDB supports indexes that include content on a single field, as well as *compound indexes* that include content from multiple fields. Continue reading for instructions and examples of building a compound index.
Build a Compound Index

To create a *compound index* use an operation that resembles the following prototype:
db.collection.ensureIndex({a:1,b:1,c:1})
The value of the field in the index specification describes the kind of index for that field. For example, a value of 1 specifies an index that orders items in ascending order. A value of -1 specifies an index that orders items in descending order. For additional index types, see *Index Types*.
Example

The following operation will create an index on the item, category, and price fields of the products collection:
db.products.ensureIndex({item:1,category:1,price:1})

Create a Unique Index

MongoDB allows you to specify a *unique constraint* on an index. These constraints prevent applications from inserting *documents* that have duplicate values for the inserted fields. Additionally, if you want to

create an index on a collection that has existing data that might have duplicate values for the indexed field, you may choose to combine unique enforcement with *duplicate dropping*.
Unique Indexes
To create a *unique index*, consider the following prototype:
db.collection.ensureIndex({a:1},{unique:true})
For example, you may want to create a unique index on the "tax-id": of the accounts collection to prevent storing multiple account records for the same legal entity:
db.accounts.ensureIndex({"tax-id":1},{unique:true})
The *_id index* is a unique index. In some situations you may consider using the _id field itself for this kind of data rather than using a unique index on another field.
In many situations you will want to combine the unique constraint with the sparse option. When MongoDB indexes a field, if a document does not have a value for a field, the index entry for that item will be null. Since unique indexes cannot have duplicate values for a field, without the

sparse option, MongoDB will reject the second document and all subsequent documents without the indexed field. Consider the following prototype.
db.collection.ensureIndex({a:1},{unique:true,sparse:true})
You can also enforce a unique constraint on *compound indexes*, as in the following prototype:
db.collection.ensureIndex({a:1,b:1},{unique:true})
These indexes enforce uniqueness for the *combination* of index keys and *not* for either key individually.

Drop Duplicates
To force the creation of a *unique index*index on a collection with duplicate values in the field you are indexing you can use the dropDups option. This will force MongoDB to create a *unique* index by deleting documents with duplicate values when building the index. Consider the following prototype invocation of ensureIndex():
db.collection.ensureIndex({a:1},{unique:true,dropDups:true})

Create a Sparse Index
Sparse indexes are like non-sparse indexes, except that they omit references to documents that do not include the indexed field. For fields that are only present in some documents sparse indexes may provide a significant space savings.
Prototype
To create a *sparse index* on a field, use an operation that resembles the following prototype:
db.collection.ensureIndex({a:1},{sparse:true})
Example
The following operation, creates a sparse index on the users collection that *only* includes a document in the index if the twitter_name field exists in a document.
db.users.ensureIndex({twitter_name:1},{sparse:true})
The index excludes all documents that do not include the twitter_name field.
Create a Hashed Index
To create a hashed index, specify hashed as the value of the index key, as in the following example:
Example
Specify a hashed index on _id
db.collection.ensureIndex({ _id: "hashed" })

Conclusion : Successfully implemented indexing with suitable example using MongoDB.