

A

LABORATORY MANAUL

On

**“ELECTIVE – I – ADVANCED DATABSE MANAGEMENT SYSTEM”**

**SEMESTER – (I)**

**NAME OF LABORATORY: SOFTWARE LABORATORY-I**

**DEPARTMENT OF INFORMATION AND TECHNOLOGY**

**Maratha Vidya Prasarak Samaj’s Karmaveer Adv. Baburao Ganpatrao Thakare College of Engineering Nashik**

**A.Y. 2023-24**

**Department of Information Technology, KBTCOE**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **SR. NO.** | **TITLE** | **PAGE NO.** | **DATE** | **SIGN** |
|  | **Group C: MongoDB** |  |  |  |
| 1. | Create a database with suitable example using MongoDB and implement   * 1. Inserting and saving document (batch insert, insert validation)   2. Removing document   3. Updating document (document replacement, using modifiers, upserts, updating multiple documents, returning updated documents)   4. 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) |  |  |  |
| 2. | Execute at least 10 queries on any suitable MongoDB database that demonstrates following querying techniques:  1. find and findOne (specific values)Query criteria (Query conditionals, OR queries, $not, Conditional semantics)  2. Type-specific queries (Null, Regular expression, Querying arrays) |  |  |  |
| 3. | Execute at least 10 queries on any suitable MongoDB database that demonstrates following:  1. $ where queries   * 1. Cursors (Limits, skips, sorts, advanced query options)   2. Database commands |  |  |  |
| 4. | Implement Map reduce example with suitable example. |  |  |  |
| 5. | Implement the aggregation and indexing with suitable example in MongoDB. Demonstrate the following:   * 1. Aggregation framework   2. Create and drop different types of indexes and explain to show the advantage of the indexes. |  |  |  |
|  | **Group D: Mini Project / Database Application Development** |  |  |  |

**Experiment No: Group\_C\_01 Date:**

**1.1 Aim:**  Create a database with suitable example using MongoDB and implement

1. Inserting and saving document (batch insert, insert validation)
2. Removing document
3. Updating document (document replacement, using modifiers, upserts, updating multiple documents, returning updated documents)

**1.2 Objectives:** 1. To learn the concept of MongoDB.

2. Learn to access the data from MongoDB.

**1.3 Hardware used:** Computer System

**1.4 Softwares used / Programming Languages Used:**

Open source operating system (Linux).

MongoDB.

**1.5. Theory:**

**Introduction:**

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. To create a database in MongoDB use following commands:

**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

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

>show dbs

local 0.78125GB

test 0.23012GB

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

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

>show dbs

local 0.78125GB

mydb 0.23012GB

test 0.23012GB

**The drop Database() 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 of 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 }

Now check list of databases.

>show dbs

local 0.78125GB

test 0.23012GB

**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 is used to specify configuration of collection.

**The insert() Method**

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

**Syntax**

The 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: 'tutorials point',

url: 'http://www.tutorialspoint.com',

tags: ['mongodb', 'database', 'NoSQL'],

likes: 100

})

**The find() Method**

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

**Syntax**

The 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()

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

{

"\_id": ObjectId(7df78ad8902c),

"title": "MongoDB Overview",

"description": "MongoDB is no sql database",

"by": "tutorials point",

"url": "http://www.tutorialspoint.com",

"tags": ["mongodb", "database", "NoSQL"],

"likes": "100"

}

**The remove () Method**

MongoDB's **remove ()** method is used to remove a 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)

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

**Syntax**

The basic syntax of **update()** method is as follows −

>db.COLLECTION\_NAME.update(SELECTION\_CRITERIA,

UPDATED\_DATA)

**1.6 Conclusion:** In this way we can study various SQL commands. All these commands (DDL, DCL and DML) are used to access the database information.

**1.7 Questions**

1.What is MongoDB?

2. Enlist the features of MongoDB?

3. What are different components of MongoDB?

**Signature of Staff with Date**

**Experiment No: Group\_C\_02 Date:**

**1.1 Aim:**  Execute at least 10 queries on any suitable MongoDB database that demonstrates following querying techniques:

1. find and findOne (specific values)
2. Query criteria (Query conditionals, OR queries, $not, Conditional semantics)
3. Type-specific queries (Null, Regular expression, Querying arrays).

**1.2 Objectives: 1.** To learn various MongoDB operations to access data from mongo

Database.

**1.3 Hardware used:** Computer System

**1.4 Softwares used / Programming Languages Used:**

**** Open source operating system (Linux).

MongoDB.

**1.5. Theory:**

**Introduction:**

Mongodb gives different operations to access data from mongo database. Following are some of the operations.

**db.collection.find()**

db.collection.find (*query*, *projection*)

Selects documents in a collection and returns a [cursor](https://docs.mongodb.com/manual/reference/glossary/#term-cursor) to the selected documents.

**Parameter Type Description**

Query document Optional. Specifies selection filter using [query](https://docs.mongodb.com/manual/reference/operator/)

[operators](https://docs.mongodb.com/manual/reference/operator/). To return all documents in a collection,

omit this parameter or pass an empty document.

Projection document Optional. Specifies the fields to return in the

documents that match the query filter. To return all

fields in the matching documents, omit this parameter.

**db.collection.findOne()**

db.collection.findOne(*query*, *projection*)

This returns one document that satisfies the specified query criteria. If multiple documents satisfy the query, this method returns the first document according to the [natural order](https://docs.mongodb.com/manual/reference/glossary/#term-natural-order) which reflects the order of documents on the disk. In [capped collections](https://docs.mongodb.com/manual/reference/glossary/#term-capped-collection),

natural order is the same as insertion order. If no document satisfies the query, the method returns null.

**Parameter Type Description**

query document Optional. Specifies query selection criteria using

query operators.projection document Optional. Specifies the fields to return using

 projection operators. Omit this parameter to return

all fields in the matching document.

**$OR**

The $or operator performs a logical OR operation on an array of two or more <expressions> and selects the documents that satisfy at least one of the <expressions>. The $or has the following syntax:

{ $or: [ { <expression1> }, { <expression2> }, ... , { <expressionN> } ] }

Consider the following example:

db.inventory.find( { $or: [ { quantity: { $lt: 20 } }, { price: 10 } ] } )

This query will select all documents in the inventory collection where either the quantity field value is less than 20 **or** the price field value equals 10.

**$NOT**

Syntax: { field: { $not: { <operator-expression> } } }

$not performs a logical NOT operation on the specified <operator-expression> and selects the documents that do not match the <operator-expression>. This includes documents that do not contain the field.

Consider the following query:

db.inventory.find( { price: { $not: { $gt: 1.99 } } } )

This query will select all documents in the inventory collection where:

the price field value is less than or equal to 1.99 **or**

the price field does not exist

{ $not: { $gt: 1.99 } } is different from the [$lte](https://docs.mongodb.com/manual/reference/operator/query/lte/#op._S_lte) operator. {$lte: 1.99} returns only the documents where price field exists and its value is less than or equal to 1.99.

**Query for Null or Missing Fields**

Different query operators in MongoDB treat null values differently.

The examples is given below,

db.users.insert(

[{ "\_id" : 900, "name" : null },

{ "\_id" : 901 }

 ])

The { name : null } query matches documents that either contain the name field whose value is null or that do not contain the name field.

Given the following query:

db.users.find( { name: null } )

The query returns both documents:

{ "\_id" : 900, "name" : null }

{ "\_id" : 901 }

**Regular Expression**

MongoDB provides functionality of regular expression for string pattern matching using the **$regex** operator. MongoDB uses PCRE (Perl Compatible Regular Expression) as regular expression language. Unlike text search, we do not need to do any configuration or command to use regular expressions.

Consider the following document structure under **posts** collection containing the post text and its tags −

{

"post\_text": "enjoy the mongodb articles on tutorialspoint",

"tags": [

"mongodb",

"tutorialspoint"

]}

**Using regex Expression**

The following regex query searches for all the posts containing string **tutorialspoint** in it −

>db.posts.find({post\_text:{$regex:"tutorialspoint"}})

The same query can also be written as −

>db.posts.find({post\_text:/tutorialspoint/})

**The pretty() Method**

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

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

**MongoDB Update() Method**

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

The basic syntax of **update()** method is as follows −

>db.COLLECTION\_NAME.update(SELECTION\_CRITERIA, UPDATED\_DATA)

**MongoDB Save() Method**

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

The basic syntax of MongoDB **save()** method is shown below −

>db.COLLECTION\_NAME.save({\_id:ObjectId(),NEW\_DATA})

**1.6 Conclusion:** In this way we can use various mongodb methods to perform queries.

**1.7 Questions**

1.What is Mongodb?

2. Explain find method with suitable example?

**Signature of Staff with Date**

**Experiment No: Group\_C\_03 Date:**

**1.1 Aim:** Execute at least 10 queries on any suitable MongoDB database that demonstrates following:

$ where queries

Cursors (Limits, skips, sorts, advanced query options)

Database commands

**1.2 Objectives:** 1. To learn SQL DCL, DDL commands.

2. To Learn ER diagram and its features.

**1.3 Hardware used:** Computer System

**1.4 Softwares used / Programming Languages Used:**

Open source operating system (Linux).

MongoDB

**1.5. Theory:**

**Introduction:**

**Where**

We can use the $where operator to pass either a string containing a JavaScript expression or a full JavaScript function to the query system. The $where provides greater flexibility, but requires that the database processes the JavaScript expression or function for each document in the collection. Reference the document in the JavaScript expression or function using either this or obj .

Examples:

db.myCollection.find( { $where: "this.credits == this.debits" } );

db.myCollection.find( { $where: "obj.credits == obj.debits" } );

db.myCollection.find( { $where: function() { return (this.credits = this.debits) } } );

db.myCollection.find( { $where: function() { return obj.credits == obj.debits; } } );

**Cursor**

The [db.collection.find()](https://docs.mongodb.com/manual/reference/method/db.collection.find/#db.collection.find) method returns a cursor. To access the documents, we need to iterate the cursor. However, in the [mongo](https://docs.mongodb.com/manual/reference/program/mongo/#bin.mongo) shell, if the returned cursor is not assigned to a variable using the var keyword, then the cursor is automatically iterated up to 20 times to print up to the first 20 documents in the results. In the [mongo](https://docs.mongodb.com/manual/reference/program/mongo/#bin.mongo) shell, when we

assign the cursor returned from the [find()](https://docs.mongodb.com/manual/reference/method/db.collection.find/#db.collection.find) method to a variable using the var keyword, the cursor does not automatically iterate. We can call the cursor variable in the shell

to iterate up to 20 times and print the matching documents, as in the following example:

var myCursor = db.users.find( { type: 2 } );

myCursor

You can also use the cursor method [next()](https://docs.mongodb.com/manual/reference/method/cursor.next/#cursor.next) to access the documents, as in the below example:

var myCursor = db.users.find( { type: 2 } );

while (myCursor.hasNext()) {

print(tojson(myCursor.next()));

}

As an alternative print operation, consider the printjson() helper method to replace print(tojson()):

var myCursor = db.users.find( { type: 2 } );

while (myCursor.hasNext()) {

printjson(myCursor.next());

}

We can use the cursor method [for Each()](https://docs.mongodb.com/manual/reference/method/cursor.forEach/#cursor.forEach) to iterate the cursor and access the documents, as in the following example:

var myCursor = db.users.find( { type: 2 } );

myCursor.forEach(printjson);

**cursor.limit()**

We can use the limit() method on a cursor to specify the maximum number of documents the cursor will return. limit() is analogous to the LIMIT statement in a SQL database. Use limit() to maximize performance and prevent MongoDB from returning more results than required for processing.

The cursor.limit () method has the following prototype form:

db.collection.find (<query>).limit(<number>)

**skip (aggregation)**

We can skips over the specified number of [documents](https://docs.mongodb.com/manual/reference/glossary/#term-document) that pass into the stage and passes the remaining documents to the next stage in the [pipeline](https://docs.mongodb.com/manual/reference/glossary/#term-pipeline).

The $skip stage has the following prototype form:

{ $skip: <positive integer> }

$skip takes a positive integer that specifies the maximum number of documents to skip.

**Example**

Consider the following example:

db.article.aggregate(

{ $skip : 5 }

);

**cursor.skip()**

We can call the cursor.skip() method on a cursor to control where MongoDB begins returning results. This approach may be useful in implementing “paged” results.

(We must apply cursor.skip() to the cursor before retrieving any documents from the database.)

Consider the following JavaScript function as an example of the skip function:

function printStudents(pageNumber, nPerPage) {

print("Page: " + pageNumber);

db.students.find().skip(pageNumber > 0 ? ((pageNumber-1)\*nPerPage) : 0).limit(nPerPage).forEach( function(student) { print(student.name + "<p>"); } );

}

The cursor.skip() method is often expensive because it requires the server to walk from the beginning of the collection or index to get the offset or skip position before beginning to return results. As the offset (e.g. pageNumber above) increases, cursor.skip() will become slower and more CPU intensive. With larger collections, cursor.skip() may become IO bound.

**cursor.sort()**

It specifies the order in which the query returns matching documents. We must apply sort() to the cursor before retrieving any documents from the database.

The sort() method has the following parameter:

| **Parameter** | **Type** | **Description** |
| --- | --- | --- |
| sort | document | A document that defines the sort order of the result set. |

The sort parameter contains field and value pairs, in the following form:

{ field: value }

**Result Ordering**

Unless we specify the sort() method or use the [$near](https://docs.mongodb.com/manual/reference/operator/query/near/#op._S_near) operator, MongoDB does not ****guarantee the order of query results.

**Ascending/Descending Sort**

Specify in the sort parameter the field or fields to sort by and a value of 1 or -1 to specify an ascending or descending sort respectively. The following sample document specifies a descending sort by the age field and then an ascending sort by the posts field:

{ age : -1, posts: 1 }

When comparing values of different [BSON types](https://docs.mongodb.com/manual/reference/bson-types/#bson-types), MongoDB uses the following comparison order, from lowest to highest:

1. MinKey (internal type)
2. Null
3. Numbers (ints, longs, doubles, decimals)
4. Symbol, String
5. Object
6. Array
7. BinData
8. ObjectId
9. Boolean
10. Date
11. Timestamp
12. Regular Expression
13. MaxKey (internal type)

**Database Commands**

All command documentation given below describes a command and its available parameters and provides a document template or prototype for each command. Some command documentation also includes the relevant [mongo](https://docs.mongodb.com/manual/reference/program/mongo/#bin.mongo) shell helpers.

To run a command, use the [db.runCommand()](https://docs.mongodb.com/manual/reference/method/db.runCommand/#db.runCommand):

db.runCommand( { <command> } )

**User Commands**

**Aggregation Commands**

**Name Description**

[aggregate](https://docs.mongodb.com/manual/reference/command/aggregate/#dbcmd.aggregate) Performs [aggregation tasks](https://docs.mongodb.com/manual/core/aggregation-pipeline/) such as group using the aggregation

framework.

[count](https://docs.mongodb.com/manual/reference/command/count/#dbcmd.count) Counts the number of documents in a collection.

[distinct](https://docs.mongodb.com/manual/reference/command/distinct/#dbcmd.distinct) Displays the distinct values found for a specified key in a

collection.

[group](https://docs.mongodb.com/manual/reference/command/group/#dbcmd.group) *Deprecated*. Groups documents in a collection by the specified

key and performs simple aggregation.

[mapReduce](https://docs.mongodb.com/manual/reference/command/mapReduce/#dbcmd.mapReduce) Performs [map-reduce](https://docs.mongodb.com/manual/core/map-reduce/) aggregation for large data sets.

**Geospatial Commands**

**Name Description**

[geoNear](https://docs.mongodb.com/manual/reference/command/geoNear/#dbcmd.geoNear) Performs a geospatial query that returns the documents closest

to a given point.

[geoSearch](https://docs.mongodb.com/manual/reference/command/geoSearch/#dbcmd.geoSearch) Performs a geospatial query that uses MongoDB’s [haystack](https://docs.mongodb.com/manual/reference/glossary/" \l "term-haystack-index)

[index](https://docs.mongodb.com/manual/reference/glossary/" \l "term-haystack-index) functionality

**Query and Write Operation Commands**

**Name Description**

Find Selects documents in a collection.

Insert Inserts one or more documents.

Update Updates one or more documents.

Delete Deletes one or more documents.

findAndModify Returns and modifies a single document.

getMore Returns batches of documents currently pointed to by the cursor.

getLastError Returns the success status of the last operation.

getPrevError Returns status document containing all errors since the last

resetError command.resetError Resets the last error status.

Eval Deprecated. Runs a JavaScript function on the database server.

parallelCollectionScan Lets applications use multiple parallel cursors when

reading documents from a collection.

****

**1.6 Conclusion:** In this way we can study cursors in mongodb. Cursors can be applied to databases for specific purposes. We can also use various database commands for different operations.

**1.7 Questions**

1.What are where queries in mongodb?

2. What are cursors?

3. What are different database commands?

**Signature of Staff with Date**

**Experiment No: Group\_C\_04 Date:**

**1.1 Aim:** Implement Map reduce example with suitable example.

**1.2 Objectives:** 1. To learn the map reduce algorithm.

2. Learn to apply map reduce algorithm for different Applications.

**1.3 Hardware used:** Computer System

**1.4 Softwares used / Programming Languages Used:**

Open source operating system (Linux).

 MongoDB

**1.5. Theory:**

**Introduction:**

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

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

**Using MapReduce**

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

{

"post\_text": "tutorialspoint is an awesome website for tutorials",

"user\_name": "mark",

"status":"active"

}

For example, use a map Reduce function on our posts collection to select all the active posts, group them on the basis of user\_name and then count the number of posts by each user using the following code −

>db.posts.mapReduce(

function() { emit(this.user\_id,1); },

function(key, values) {return Array.sum(values)}, {

query:{status:"active"},

out:"post\_total"

}

)

The above map Reduce query outputs the following result −

{

"result" : "post\_total",

"timeMillis" : 9,

"counts" : {

"input" : 4,

"emit" : 4,

"reduce" : 2,

"output" : 2

},

"ok" : 1,

}

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

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

>db.posts.mapReduce(

function() { emit(this.user\_id,1); },

function(key, values) {return Array.sum(values)}, {

query:{status:"active"},

out:"post\_total"

}

).find()

The above query gives the following result which indicates that both user’s tom and mark have two posts in active states-

{ "\_id" : "tom", "value" : 2 }

{ "\_id" : "mark", "value" : 2 }

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

**1.6 Conclusion:** In this way we can study MapReduce function for handling large data.

**1.7 Questions**

1.What is BigData?

2. What MapReduce function?

**Signature of Staff with Date**

**Experiment No: Group\_C\_05 Date:**

**1.1 Aim:** Implement the aggregation and indexing with suitable example in MongoDB. Demonstrate the following:

1. Aggregation framework
2. Create and drop different types of indexes and explain () to show the advantage of the indexes.

**1.2 Objectives:** 1. To learn the aggregation and indexing concept in mongodb.

2. To Learn the concept of index in mongodb.

**1.3 Hardware used:** Computer System

**1.4 Softwares used / Programming Languages Used:**

Open source operating system (Linux).

MongoDB

**1.5. Theory:**

**Introduction:**

**Aggregation**

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

**The 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)

**Example**

In the collection you have the following data −

{

\_id: ObjectId(7df78ad8902c)

title: 'MongoDB Overview',

description: 'MongoDB is no sql database',

by\_user: 'tutorials point',

url: 'http://www.tutorialspoint.com',

tags: ['mongodb', 'database', 'NoSQL'],

likes: 100

},

{

\_id: ObjectId(7df78ad8902d)

title: 'NoSQL Overview',

 by\_user: 'tutorials point',

url: 'http://www.tutorialspoint.com',

tags: ['mongodb', 'database', 'NoSQL'],

likes: 10

},

{

\_id: ObjectId(7df78ad8902e)

title: 'Neo4j Overview',

description: 'Neo4j is no sql database',

by\_user: 'Neo4j',

url: 'http://www.neo4j.com',

tags: ['neo4j', 'database', 'NoSQL'],

likes: 750

},

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

> db.mycol.aggregate([{$group : {\_id : "$by\_user", num\_tutorial : {$sum : 1}}}])

{

"result" : [

{

"\_id" : "tutorials point",

"num\_tutorial" : 2

},

{

"\_id" : "Neo4j",

"num\_tutorial" : 1

}

],

"ok" : 1

}

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

In the above example, we have grouped documents by field **by\_user** and on each occurrence of by\_user previous value of sum is incremented.

**Indexing**

Indexes support the efficient resolution of queries. Without indexes, MongoDB must scan every document of a collection to select those documents that match the query statement. This scan is highly inefficient and requires MongoDB to process a large volume of data. Indexes are special data structures, that store a small portion of the data set in an easy-to-traverse form. The index stores the value of a specific field or set of fields, ordered by the value of the field as specified in the index.

**The ensureIndex() Method**

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

**Syntax**

The basic syntax of **ensureIndex()** method is as follows().

>db.COLLECTION\_NAME.ensureIndex({KEY:1})

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

**Example**

>db.mycol.ensureIndex({"title":1})

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

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

**Index Creation**

MongoDB provides several options to create indexes. By default, when indexes are created, all other operations on a database are blocked. For example, when indexes on

a collection are created, the database becomes unavailable for any read or writes operation until the index creation process completes. The read or write operations on

the database queue and allow the index building process to complete. Therefore, for index building operations which may consume longer time, we can consider the background operation and thus make MongoDB available even during the entire operation. The command given on the screen is used for this purpose. By default, background is false for building MongoDB indexes.

When MongoDB is creating indexes in the background for a collection, we cannot perform other administrative operations involving that collection. For example, we cannot perform tasks, such as runrepair Database, (read as run repair database) drop the collection, or use the query db.collection.drop(),(read as D-B dot collection dot drop) and runcompact (read as run compact). If we perform any of these operations, we will receive an error. The index build process at the background uses an incremental approach and is slower than the normal “foreground” index build process. The speed of the index build process depends on the size of the index. If the index size is bigger than the RAM of the system, the process takes more time than the foreground process. Building indexes can impact database performance: • If the application includes createIndex()(read as create index) operations and • If no index is available for operational concerns. To avoid any performance issues, we can use the getIndexes()(read as det indexes) method to ensure that your application checks for the indexes at the start up. We can also use an equivalent method for your driver and ensure it terminates an operation if the proper indexes do not exist. When building indexes, use separate application codes and designated maintenance windows.

**Remove Indexes**

We can use the following methods to remove indexes.

dropIndex() (read as drop index) method: This removes an index from a collection. db.collection.dropIndex() (read as D-B dot collection dot drop index) method: This removes an index. For example, the first operation given on the screen removes an ascending index on the item field in the items collection. To remove all indexes barring the \_id index from a collection, use the second operation provided on the screen.

**1.6 Conclusion:** In this way we can the concept of aggregation and indexing in mongodb.

**1.7 Questions**

 1.What is aggregation in mongodb?

2. What is indexing in mongodb?

**Signature of Staff with Date**

**Experiment No: Group\_D\_01 Date:**

**1.1 Aim:** To implement a mini project/sample application.

**1.2 Objectives:** Study and implement simple commercial application using PHP as Front End and MongoDB as Back End.

**1.3 Hardware used:** Computer System

**1.4 Softwares used / Programming Languages Used:**

Open source operating system (Linux).

 MongoDB

**1.5 Theory:**

Prepare a detailed report of your mini project covering following details:

1. Title of the Project.

2. Abstract

3. Introduction

4. Scope

5. Software/Hardware Requirements Specification

6. Entity Relationship Diagram with EER features

7. Data Dictionary

8. Relational Database Design

9. Database Normalization

10. Graphical User Interface(Screenshots)

11. Source Code: (should be in CD)

12. Data Reports

13. Testing document

14. Future Enhancement

15. Conclusion

16. References/Bibliography