**Part C- MongoDB Writeup**

**Assignment No. 1**

**Aim: -:** Create a database with suitable example using MongoDB and implement Inserting,updating, removing and saving document

**Objective:** Perform CURD operation on MongoDB Database

**Theory:**

**What is MongoDB**

MongoDB is an open-source document database that provides high performance, high availability, and automatic scaling.

**Document Database**

A record in MongoDB is a document, which is a data structure composed of field and value

pairs. MongoDB documents are similar to JSON objects. The values of fields may include other

documents, arrays, and arrays of documents.

Figure shows a MongoDB document.



The advantages of using documents are:

* Documents (i.e. objects) correspond to native data types in many programming languages.
* Embedded documents and arrays reduce need for expensive joins.
* Dynamic schema supports fluent polymorphism.

**Key Features**

**High Performance**

MongoDB provides high performance data persistence. In particular,• Support for embedded data models reduces I/O activity on database system.

• Indexes support faster queries and can include keys from embedded documents and arrays.

**High Availability**

To provide high availability, MongoDB’s replication facility, called replica sets, provide:

* Automatic failover.
* Data redundancy.

A replica set is a group of MongoDB servers that maintain the same data set, providing redundancy and increasing data availability.

**Automatic Scaling**

MongoDB provides horizontal scalability as part of its core functionality.

* Automatic sharding distributes data across a cluster of machines.
* Replica sets can provide eventually-consistent reads for low-latency high throughput deployments.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **SQL Vs MongoDB** | | | | | | | | | |  |  |  |  | | | | | | | | | |
|  |  | | | | | | | | |  |  |  |  | | | | | | | | | |
|  | **SQL Concepts** | | | | | | | | |  |  |  | **MongoDB Concepts** | | | | | | | | | |
|  |  | | | | | | | | |  |  |  |  | | | | | | | | | |
|  |  | | | | | | | | |  |  |  |  | | | | | | | | | |
|  |  | | | | | | | | |  |  |  |  | | | | | | | | | |
|  | database | | | | | | | | |  | database | | | | | | | | | | | |
|  |  | | | | | | | | |  |  |  |  | | | | | | | | | |
|  |  | | | | | | | | |  |  |  |  | | | | | | | | | |
|  | table | | | | | | | | |  | Collection | | | | | | | | | | | |
|  |  | | | | | | | | |  |  |  |  | | | | | | | | | |
|  |  | | | | | | | | |  |  |  |  | | | | | | | | | |
|  | Row | | | | | | | | |  | Document 0r BSON Document | | | | | | | | | | | |
|  |  | | | | | | | | |  |  |  |  | | | | | | | | | |
|  |  | | | | | | | | |  |  |  |  | | | | | | | | | |
|  |  | | | | | | | | |  |  |  |  | | | | | | | | | |
|  | Column | | | | | | | | |  | Field | | | | | | | | | | | |
|  |  | | | | | | | | |  |  |  |  | | | | | | | | | |
|  | Index | | | | | | | | |  | Index | | | | | | | | | | | |
|  |  | | | | | | | | |  |  |  |  | | | | | | | | | |
|  | Table Join | | | | | | | | |  | Embedded documents & Linking | | | | | | | | | | | |
|  |  | | | | | | | | |  |  |  |  | | | | | | | | | |
|  |  | | | | | | | | |  |  |  |  | | | | | | | | | |
|  | Primary key | | | | | | | | |  | Primary Key | | | | | | | | | | | |
|  |  | | | | | | | | |  |  |  |  | | | | | | | | | |
|  |  | | | | | | | | |  |  |  |  | | | | | | | | | |
|  | Specify any unique column or column | | | | | | | | |  | In MongoDB, the primary key is automatically | | | | | | | | | | | |
|  | combination as primary key. | | | | | | | | |  | set to the [*\_id*](http://docs.mongodb.org/manual/reference/glossary/) field. | | | | | | | | | | | |
|  |  | | | | | | | | |  |  |  |  | | | | | | | | | |
|  |  | | | | | | | | |  |  |  |  | | | | | | | | | |
|  | aggregation (e.g. group by) | | | | | | | | |  | aggregation pipeline | | | | | | | | | | | |
|  |  | | | | | | | | |  |  |  |  | | | | | | | | | |
|  |  | | | | | | | | |  |  |  |  | | | | | | | | | |
|  |  | **Executables** | | | |  |  |  |  |  |  |  | | | | |  |  |  |  |  |  |  |  |
|  |  |  | | |  |  |  |  |  |  |  |  | | | | |  |  |  |  |  |  |  |  |
|  |  |  | | |  |  |  | **Oracle** |  |  |  | **MySQL** | | | | |  |  |  | **MongoDB** | | |  |  |
|  |  |  | | |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  | | |  |  |  |  |  |  |  |  | | | | |  |  |  |  |  |  |  |  |
|  |  |  | | |  |  |  |  |  |  |  |  | | | | |  |  |  |  |  |  |  |  |
|  |  | | | | |  |  |  | |  |  |  | | | | | |  |  |  |  |  | |  |
|  |  | | | | |  |  |  | |  |  |  | | | | | |  |  |  |  |  | |  |
|  |  | Database Server | | |  |  |  | oracle |  |  |  | mysqld | | | | |  |  |  | [mongod](http://docs.mongodb.org/manual/reference/program/mongod/) | | |  |  |
|  |  |  | | |  |  |  |  |  |  |  |  | | | | |  |  |  |  |  |  |  |  |
|  |  |  | | |  |  |  |  |  |  |  |  | | | | |  |  |  |  |  |  |  |  |
|  |  |  | | |  |  |  |  |  |  |  |  | | | | |  |  |  |  |  | | |  |
|  |  | Database Client | | |  |  |  | sqlplus |  |  |  | mysql | | | | |  |  |  | [mongo](http://docs.mongodb.org/manual/reference/program/mongo/) | | |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  | | |  |  |  |  |  |  |  |  | | | | |  |  |  |  |  |  |  |  |

|  |  |
| --- | --- |
| **Purpose** | **MongoDB query** |
| Create collection=table | db.createCollection("stud") |
| Insert a document/record  ***insert()***  **Returns**:   * A **WriteResult** object or single inserts * A **BulkWriteResult** object for bulk inserts | db.stud.insert( {name:"a", age:2, status:"B" })  *WriteResult({ "nInserted" : 1 })* |
| Insert multiple documents | db.products.insert(  [  { item: "pencil", qty: 50, type: "no.2" },  { item: "pen", qty: 20 },  { item: "eraser", qty: 25 }  ]  )  *BulkWriteResult({*  *"writeErrors" : [ ],*  *"writeConcernErrors" : [ ],*  *"nInserted" : 3,*  *"nUpserted" : 0,*  *"nMatched" : 0,*  *"nModified" : 0,*  *"nRemoved" : 0,*  *"upserted" : [ ]*  *})* |
| Ordered and Unordered insert:  db.products.insert(  [  { \_id: 20, item: "lamp", qty: 50, type: "desk" },  { \_id: 21, item: "lamp", qty: 20, type: "floor" },  { \_id: 22, item: "bulk", qty: 100 }  ],  **{ ordered: false }**  )  If true, perform an ordered insert of the documents in the array, and if an error occurs with one of documents, MongoDB will return without processing the remaining documents in the array.  If false, perform an unordered insert, and if an error occurs with one of documents, continue processing the remaining documents in the array.  **Default value- true** | |
| InsertOne  (Allows to insert exactly 1 document in the collection) | db.products.insertOne( { item: "card", qty: 15 } );  *output:*  *{*  *"acknowledged" : true,*  *"insertedId" : ObjectId("571a218011a82a1d94c02333")*  *}* |
| Insert Many | db.stud.insertMany(  [  { name: "bob", age: 42, status: "A" },  { name: "ahn", age: 22, status: "A" },  ]  )  *output:*  *{*  *"acknowledged" : true,*  *"insertedIds" : [*  *ObjectId("571a22a911a82a1d94c02337"),*  *ObjectId("571a22a911a82a1d94c02338")*  *]*  *}* |
| Update  *syntax :* db.collection.update(query, update, options)  Query - The selection criteria for the update.  Update - The modifications to apply. | db.stud.update(  { name: "bob" }, { $set: { age:50 } }  ) |
| Updates multiple rows/documents  multi:Optional. If set to true, updates multiple documents that meet the query criteria. If set to false, updates one document. The default value is false. | db.user.update(  { age:{$gt:18 }}, {$set:{status:"a"}},  {multi:true}  ) |
| upsert:Optional. If set to true, creates a new document when no document matches the query criteria. The default value is false, which does not insert a new document when no match is found. | db.people.update(  { age:{$gt:18 }}, {$set:{status:"a"}},  { upsert: true }  ) |
| updateOne | db.users.updateOne(  { "user\_id": "abc12" },  {  $set: { "age": 20},  $currentDate: { lastModified: true }  }  ) |
| updateMany | db.users.updateMany(  { "status": "a" },  {  $set: { "user\_id": "vaishu", type: 3 },  $currentDate: { lastModified: true }  }  ) |
| replaceOne | db.users.replaceOne(  { name: "Sally" },  { name: "amy", age: 34, type: 2 }    )  *//type gets added at the end of the document /row*  *//output is{ "acknowledged" : true, "matchedCount" : 0, "modifiedCount" : 0 as name field not in document. so change to* |
| replaceOne | db.users.replaceOne(  { name: "Sally" },  { name: "amy", age: 34, type: 2 } ,  {upsert:true}  ) |
| findOneAndReplace | db.users.findOneAndReplace(  { "user\_id" : "Sally" },  { "user\_id":"Amy","age" : 28,"type":22},  { upsert : true, returnNewDocument: true }  )  //Displays the updated document |
| Document Validation | db.createCollection("customer", {  validator: {  email: { $exists: true }  }  })  **Valid Example:**  db.customer.insert({\_id: 1, email:'xyz@gmail.com'})  db.customer.insert({\_id: 4, email: 42})  **Invalid Example:**  db.customer.insert({\_id: 2, name: 'iggy'})  db.customer.insert({\_id: 3, emailID: 'xyz@gmail.com'}) |
| Document Validation | db.createCollection("address",  {  validator:{  firstname:{$type:"string"},  lastname:{$type:"string"},  emailid:{$type:"string",$regex:/@TutorialToUs\.com/},  country:{$in:["UK","INDIA"]},  pincode:{$type:"string",$regex:/...../}  }  }) |
| Document Validation | db.createCollection("contacts",  {      validator: { $or:            [                { phone: { $type: "string" } },                { email: { $regex: /@testing\.com$/ } },                { status: { $in: [ "Unknown", "Incomplete" ] } }            ]        },         validationLevel: "strict",   validationAction:"error"                   } ) |
| Document Validation | db.createCollection("c",         {                  validator: {{ email: { $regex: /@mongodb\.com$/ } }},                  validationLevel: "strict",                  validationAction:"error"                          } ) |
| Document Validation | db.createCollection("testing",  {validator:  {email:{$regex: /^\w+@\w+\.+\w/}}  }  ) |
| remove | db.stud.remove( { age: { $lt: 35 } } ) |
|  | db.stud.remove({}) |
|  | db.stud.drop() |
| Rename collection | db.users.renameCollection("stud\_details") |
| show Collections | db.getCollectionNames() |
| $inc operator | *>* db.createCollection("products")  > db.products.insert({\_id: 1, sku: "abc123", quantity: 10,  metrics: { orders: 2, ratings: 3.5. } })  > db.products.update( { sku: "abc123" },  { $inc: { quantity: -2, "metrics.orders": 1 } )  *>*db.products.find();  *>{ "\_id" : 1, "sku" : "abc123", "quantity" : 8, "metrics" : { "orders" : 3, "rati*  *ngs" : 3.5 } }* |

The $inc operator increments a field by a specified value and has the following form:

{ $inc: { <field1>: <amount1>, <field2>: <amount2>, ... } }

* The $inc operator accepts positive and negative values.
* If the field does not exist, $inc creates the field and sets the field to the specified value.
* Use of the $inc operator on a field with a null value will generate an error.
* $inc is an atomic operation within a single document.

**Conclusion: -**Understand and execute MongoDB query

**Assignment No. 2**

**Aim: -:** Execute at least 10 queries on any suitable MongoDB database that demonstratesfollowing querying techniques:

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

**Objective:** Perform query operation using find( ) on MongoDB Database

**Theory:**

**Introduction to find**

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. When we start adding key/value pairs to the query document, we begin restricting our search. This works in a straightforward way for most types. Integers match integers, Booleans match Booleans, and strings match strings. 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 "joe", we use that key/value pair instead: > db.users.find({"username" : "joe"}) 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 “joe,” we can query for the following: > db.users.find({"username" : "joe", "age" : 27}) Specifying Which Keys to Return Sometimes, you do not need all of the key/value pairs in a document returned.

If this is the case, you can pass a second argument to find (**or findOne**) specifying the keys you want. This reduces both the amount of data sent over the wire and the time and memory used to decode documents on the client side.

For example, if you have a user collection and you are interested only in the "user name" and "email" keys, you could return just those keys with the following query:

* db.users.find({},{"username" : 1, "email" : 1})

|  |  |
| --- | --- |
| Regular Expressions | db.users.find( { name: { $regex: /^a/ } } ) //starting with a |
| db.users.find( { name: { $regex: /a/ } } )  // displays the docs which contains *a* character in name |
| db.users.find( { name: { $regex: /b$/ } } ) //ending with b |
| db.users.find({name:{$regex:/^s.\*m$/}}) //displays the docs starting with name s and ending with m. |
| Displaying the documents ascending/descending | db.users.find().sort(  { user\_id: 1, status: -1 }  Second key will be applied when 2 docs have user\_id values. |
| users who are between the ages of 18 and 30 inclusive, we can do this | db. users.find({"age" : {"$gte" : 18, "$lte" : 30}}) |
| Other condition | db.users.find({"username" : {"$ne" : "joe"}}) |
| db.users.find( { $or:  [ { status: "A" } , { age: 55 } ] } ) |

**Conditional semantics**

In the query, "$lt" is in the inner document; in the update, "$inc" is the key for the outer document. This generally holds true: conditionals are an inner document key, and modifiers are always a key in the outer document. Multiple conditions can be put on a single key. For example, to find all users between the ages of 20 and 30, we can query for both "$gt" and "$lt" on the "age" key:

>db.users.find({"age" : {"$lt" : 30, "$gt" : 20}})

Any number of conditionals can be used with a single key. Multiple update modifiers cannot be used on a single key, however. For example, you cannot have a modifier document such as {"$inc" : {"age" : 1}, "$set" : {age : 40}} because it modifies "age" twice. With query conditionals, no such rule applies.

**Type-Specific Queries**

MongoDB has a wide variety of types that can be used in a document. Some of these behave specially in queries. null null behaves a bit strangely. It does match itself, so if we have a collection with the following documents:

>db.c.find() { "\_id" : ObjectId("4ba0f0dfd22aa494fd523621"), "y" : null }

> db.c.find({"z" : null})

If we only want to find keys whose value is null, we can check that the key is null and exists using the "$exists" conditional: > db.c.find({"z" : {"$in" : [null], "$exists" : true}}) Unfortunately, there is no "$eq" operator, which makes this a little awkward, but "$in" with one element is equivalent.

**Regular Expressions**

Regular expressions are useful for flexible string matching.

|  |  |
| --- | --- |
| Regular Expressions | db.users.find( { name: { $regex: /^a/ } } ) //starting with a |
| db.users.find( { name: { $regex: /a/ } } )  // displays the docs which contains *a* character in name |
| db.users.find( { name: { $regex: /b$/ } } ) //ending with b |
| db.users.find({name:{$regex:/^s.\*m$/}}) //displays the docs starting with name s and ending with m. |

**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.

|  |  |
| --- | --- |
| Querying Arrays:  1.Inserting Arrays | db.stud.insert({rno:1,name:"Pooja", marks:[10,20,30]});  db.stud.insert({rno:2,name:"Pankaj", marks:[60,40,43]});  db.stud.insert({rno:3,name:"Suman",marks:[0,50,48]});  db.stud.insert({rno:4,name:"Sapna”,marks:[40,50,48]});  db.stud.insert({rno:5,name:"Dipti”,marks:[40,50,48]}); |
| 2. Finding in Arrays | db.stud.find( { marks: [40,50,48] } ) //*Exact match*  db.stud.find( { marks: {$all: [40,50,48] }} ) *// Matches arrays that contain all elements specified in the query*  db.stud.find( { marks: {$all: [40] }} ) *// $all display the output regardless the order*  db.stud.find( { marks:{$gt:40} } ) *//displays the record having marks greater than 40*  db.stud.find( { "marks.0": { $gt: 25 } } ) *//. operator used to specify the prefix or array position*  db.stud.find( { "marks.1": { $gt: 25 } } ) |

**Conclusion: -** Executed queries on MongoDB database that demonstrates querying techniques:like find and findOne, Query conditionals, OR queries, $not, Conditional semantics, Null, Regular expression, Querying arrays

**Assign No. 3**

**Aim: Execute at least 10 queries on any suitable MongoDB database to demonstrate $where, cursors and various administration commands**

**Objective**: Perform the various operations on customer database

**Theory:**

**$where Queries**

Key/value pairs are a fairly expressive way to query, but there are some queries that they cannot represent. For queries that cannot be done any other way, there are "$where" clauses, which allow you to execute arbitrary JavaScript as part of your query. This allows you to do (almost) anything within a query. 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 .

you should use "$where" only when there is no other way of doing the query. You can cut down on the penalty by using other query filters in combination with "$where". If possible, an index will be used to filter based on the non- $where clauses; the "$where" expression will be used only to fine-tune the results.

>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; } } );

Additionally, if the query consists only of the $where operator, you can pass in just the JavaScript expression or JavaScript functions, as in the following examples:

>db.myCollection.find( "this.credits == this.debits || this.credits > this.debits" );

>db.myCollection.find( **function**() { **return** (**this**.credits == **this**.debits || **this**.credits > **this**.debits ) } );

You can include both the standard MongoDB operators and the $where operator in your query, as in the following examples

>db.myCollection.find( { active: **true**, $where: "this.credits - this.debits < 0" } );

>db.myCollection.find( { active: **true**, $where: **function**() { **return** obj.credits - obj.debits < 0; } } );

**Cursors**

The database returns results from find using a cursor. The client-side implementations of cursors generally allow you to control a great deal about the eventual output of a query. You can limit the number of results, skip over some number of results, sort results by any combination of keys in any direction, and perform a number of other powerful operations.

To create a cursor with the shell, put some documents into a collection, do a query on them, and assign the results to a local variable (variables defined with "var" are local). Here, we create a very simple collection and query it, storing the results in the cursor variable:

|  |  |
| --- | --- |
| Cursors | >var myCursor1 = db.stud.find( { status:"b" } )  >myCursor1 // display the documents  The [db.collection.find()](https://docs.mongodb.com/v3.2/reference/method/db.collection.find/#db.collection.find) method returns a cursor. To access the documents, you need to iterate the cursor. However, in the [mongo](https://docs.mongodb.com/v3.2/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. |
|  | db.stud.find().forEach( function(myDoc) { print( "user: " + myDoc.name ); } );  **output:**  user: sumedh  user: abhi  user: sam  user: gon  .forEach() method Iterates the cursor to apply a JavaScript function to each document from the cursor.  //to print the name of each user in the collection:  OR  db.stud.find().map( function(u) { return u.name; } );  [ "sumedh", "abhi", "sam", "gon" ]  map() Applies function to each document visited by the cursor and collects the return values from successive application into an array. |

**Limits, Skips**

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. skip works similarly to limit:

* db.c.find().skip(3).limit(5) This will skip the first three documents and return the next 5 documents in the collection. If there are less than three documents in your collection, it will not return any documents.

**Database Commands**

MongoDB supports a wide range of advanced operations that are implemented as commands.

|  |  |
| --- | --- |
| database Commands | >use admin //switch to admin database  >db.runCommand( {buildInfo: 1} ) //*displays the information about mongodb.*  db.stud.drop()  db.createCollection()  db.orders.renameCollection( "newstud" ) |

**Conclusion: -Thus the exe**cution of commands using different queries have demonstrated

$ where queries, cursors ,Database commands

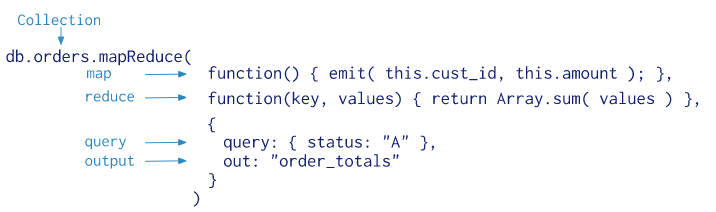
**Assignment No.4**

**Aim:** Implement MapReduce operations with suitable example using MongoDB

**Objective:** To learn MapReduce operations

**Theory:**

* MapReduce is a programming model and an associated implementation for processing and generating large data sets with a parallel, distributed algorithm on a cluster.
* A MapReduce program is composed of a Map() procedure that performs filtering and sorting (such as sorting students by first name into queues, one queue for each name) and a Reduce() procedure that performs a summary operation (such as counting the number of students in each queue, yielding name frequencies).
* 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](http://docs.mongodb.org/manual/reference/command/mapReduce/) database command.



*Draw the example shown on ppt. under the heading map-reduce*

In map Reduce we have to write 3 functions.

1. Map Function (Ex.Person to each city to count population).
   1. Reduce Function (Ex. Reducing the total population count to single value.)
   2. Map Reduce Function ( it will create a new collection it contains the total population)

**Step 1: Map**

var mapFunction1 = function() {

emit(this.cust\_id, this.price);

};

Map function to process each input document:

In the function, this refers to the document that the map-reduce operation is processing.

The function maps the price to the cust\_id for each document and emits the cust\_id and price pair.

**Step 2: Reduce**

var reduceFunction1 = function(keyCustId, valuesPrices) { return Array.sum(valuesPrices); };

Define the corresponding reduce function with two arguments keyCustId and valuesPrices: The valuesPrices is an array whose elements are the price values emitted by the map function and grouped by keyCustId. The function reduces the valuesPrice array to the sum of its elements

**Step 3: Map Reduce**

db.orders.mapReduce(

mapFunction1,

reduceFunction1,

{ out: "map\_example" }

)

Perform the map-reduce on all documents in the orders collection using the mapFunction1 map function and the reduceFunction1 reduce function. This operation outputs the results to a collection named map\_example. If the map\_example collection already exists, the operation will replace the contents with the results of this map-reduce operation:

**Map-Reduce**

MongoDB also provides map-reduce operations to perform aggregation. In general, map-reduce operations have two phases: a map stage that processes each document and emits one or more objects for each input document,and reduce phase that combines the output of the map operation. Optionally, map-reduce can have a finalize stage to make final modifications to the result. Like other aggregation operations, map-reduce can specify a query condition to select the input documents as well as sort and limit the results.

Map-reduce uses custom JavaScript functions to perform the map and reduce operations, as well as the optional finalize operation. While the custom JavaScript provide great flexibility compared to the aggregation pipeline, in general, mapreduce is less efficient and more complex than the aggregation pipeline.Additionally, map-reduce operations can have output sets that exceed the 16 megabyte output limitation of the aggregation pipeline.

**Conclusion:** Understand and implement Map Reduced Operation

**Assignment No.5**

**AIM:** Implement Aggregation and Indexing with suitable example using MongoDB.

**Objective:** To understand 1) Aggregation 2) To understand Indexing in MongoDB

**Theory:**

Indexes provide high performance read operations for frequently used queries. This section introduces indexes in MongoDB, describes the types and configuration options for indexes, and describes special types of indexing MongoDB supports. The section also provides tutorials detailing procedures and operational concerns, and providing information on how applications may use indexes. 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 1 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.

**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

**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.

|  |  |
| --- | --- |
| Indexes | db.records.createIndex({**rno:1**})  //creates an ascending index on the field rno |
| To view the name of index | >db.records.getIndexes()  Returns an array that holds a list of documents that identify and describe the existing indexes on the collection.  v: version, key: on which key the index is set, **name: indexname,** ns: namespace |
| To drop index | db.records.dropIndex("rno\_1") |
| Compound Index | db.records.createIndex( {user\_id: 1, age: -1 } )  name of index will be *user\_id\_age\_-1* |
| Multikey Index | Refer ppts…example |

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.

**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.

MongoDB provides three ways to perform aggregation: the aggregation pipeline, the map-reduce function , and single purpose aggregation methods and commands.

Aggregations are operations that process data records and return computed results. MongoDB provides a rich set of aggregation operations that examine and perform calculations on the data sets. Running data aggregation on the mongod instance simplifies application code and limits resource requirements. Like queries, aggregation operations in MongoDB use collections of documents as an input and return results in the form of one or more documents

* **SUM**
* **AVG**
* **MAX**
* **MIN**

***Use the ppt example to explain the theory***

**Conclusion:** Understand and implement various Aggregation function and Indexing