BASI DI DATI 2

DATABASE NOSQL: MONGODB

a.a 2024/2025

Prof.ssa G. Tortora

MongoDB

- MongoDB is an open-source document database and leading NoSQL database.

- MongoDB is written in C++.

- MongoDB concepts needed to create and deploy a highly scalable and performance-oriented database.

MongoDB (2)

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

Collections and Documents

- **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** 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.

Relationship of RDBMS terminology with MongoDB

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

- **_id** is a 12 bytes hexadecimal number which assures the uniqueness of every document.
  - You can provide _id while inserting the document.
  - If you don't provide it then MongoDB provides a unique id for every document.
  - 12 bytes: 4 bytes for the current timestamp, 5 bytes random value; 3 bytes incrementing counter, initialized to a random value.

Sample document

- Document structure of a blog site, which is simply a comma separated key value pair.

```
{
        _id: ObjectId(“60a4ef7e71c94e1f0d78799b”),
        title: 'MongoDB Overview’,
        description: 'MongoDB is no sql database’,
        by: ‘Basi di Dati 2’,
        url: 'http://www.unisa.it’,
        tags: ['mongodb', 'database', 'NoSQL’],
        likes: 1000,
        comments: [
                { user:'user1’,
                message: 'My first comment’,
                dateCreated: new Date(2021,1,20,2,15),
                like: 0 },
                { user:'user2’,
                message: 'My second comments’,
                dateCreated: new Date(2021,1,25,7,45),
                like: 7 }
        ]
}
```

Advantages of MongoDB over RDBMS

- **Schema less**.
  - MongoDB is a document database in which one collection holds different documents. Number of fields, content and size of the document can differ from one document to another.
- Structure of a single object is clear.
- No complex joins.
- Deep query-ability.
  - MongoDB supports dynamic queries on documents using a document-based query language that's nearly as powerful as SQL.
- Tuning.
- **Ease of scale-out**.
  - MongoDB is easy to scale.
- Conversion/mapping of application objects to database objects not needed.
- Uses internal memory for storing the (windowed) working set, enabling faster access of data.

Why Use MongoDB?

- Document Oriented Storage.
  - Data is stored in the form of JSON style documents.
- Index on any attribute
- Replication and high availability
- Auto-sharding
- Rich queries
- Fast in-place updates
- Professional support by MongoDB

Where to Use MongoDB?

- Big Data
- Content Management and Delivery
- Mobile and Social Infrastructure
- User Data Management
- Data Hub

Install MongoDB

- To install MongoDB on Windows, first download the latest release of MongoDB from https://www.mongodb.org/downloads

- You need to specify the path for **\data\db** by setting the parameter **dbpath** in **mongod**.
  - **mongod --dbpath "/data/db"**

- The **waiting for connections** message on the console output indicates that the mongod process is running successfully.

Run MongoDB (client)

- Now to run the MongoDB (client), you need to open another command prompt and issue the following command:

```
$ mongo
MongoDB shell version v4.4.4
connecting to: mongodb://127.0.0.1:27017/?gssapiServiceName=mongodb
Implicit session: session { "id" : UUID("e7c89dc7-c773-495e-9efc-281f8a261767") }
MongoDB server version: 4.4.4

> db.test.save( { a: 1} )
WriteResult({ "nInserted" : 1 })

> db.test.find()
{ "_id" : ObjectId("5cd168e0801fb24a7996247b"), "a" : 1 }
```

This server is bound to localhost
Start the server with --bind_ip <address>

MongoDB Help

- To get a list of commands, type **db.help()** in MongoDB client.
- This will give you a list of commands as shown in the following screenshot.

```
DB methods:
        db.adminCommand(nameOrDocument) - switches to 'admin' db, and runs command [just calls db.runCommand(...)]
        db.aggregate([pipeline], {options}) - performs a collectionless aggregation on this database; returns a cursor
        db.auth(username, password)
        db.cloneDatabase(fromhost) - deprecated
        db.commandHelp(name) returns the help for the command
        db.copyDatabase(fromdb, todb, fromhost) - deprecated
        db.createCollection(name, {size: ..., capped: ..., max: ...})
        db.createView(name, viewOn, [{$operator: {...}}, ...], {viewOptions})
        db.createUser(userDocument)
        db.currentOp() displays currently executing operations in the db
        db.dropDatabase()
        db.eval() - deprecated
        db.fsyncLock() flush data to disk and lock server for backups
        db.fsyncUnlock() unlocks server following a db.fsyncLock()
        db.getCollection(cname) same as db['cname'] or db.cname
        db.getCollectionInfos([filter]) - returns a list that contains the names and options of the db's collections
        db.getCollectionNames()
        db.getLastError() - just returns the err msg string
        db.getLastErrorObj() - return full status object
        db.getLogComponents()
        db.getMongo() get the server connection object
        db.getMongo().setSlaveOk() allow queries on a replication slave server
        db.getName()
        db.getPrevError()
        db.getProfilingLevel() - deprecated
        db.getProfilingStatus() - returns if profiling is on and slow threshold
        db.getReplicationInfo()
        db.getSiblingDB(name) get the db at the same server as this one
        db.getWriteConcern() - returns the write concern used for any operations on this db, inherited from server object if set
        db.hostInfo() get details about the server's host
        db.isMaster() check replica primary status
        db.killOp(opid) kills the current operation in the db
        db.listCommands() lists all the db commands
        db.loadServerScripts() loads all the scripts in db.system.js
        db.logout()
```

> use admin
switched to db admin

> db.shutdownServer()
server should be down...

> exit
bye

MongoDB Statistics

- To get stats about MongoDB server, type the command **db.stats()**.
- This will show the database name, number of collection and documents in the database.

```
{
        "db" : "test",
        "collections" : 1,
        "views" : 0,
        "objects" : 1,
        "avgObjSize" : 33,
        "dataSize" : 33,
        "storageSize" : 36864,
        "numExtents" : 0,
        "indexes" : 1,
        "indexSize" : 36864,
        "fsUsedSize" : 166895529984,
        "fsTotalSize" : 499963174912,
        "ok" : 1
}
```

Data Modelling

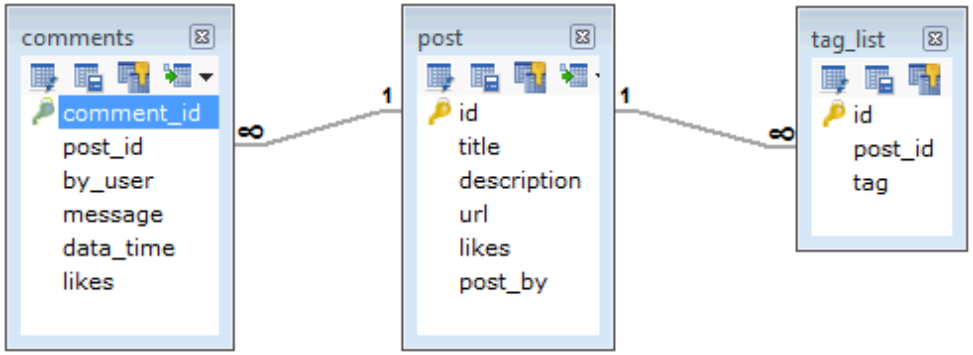
- Data in MongoDB has a flexible *schema.documents* in the same collection.
- They do not need to have the same set of fields or structure, and common fields in a collection's documents may hold different types of data.
  - Design your schema according to user requirements.
  - Combine objects into one document if you will use them together.
    - Otherwise separate them (but make sure there should not be need of joins).
  - Duplicate the data (but limited) because disk space is cheap as compare to compute time.
  - Do joins while write, not on read.
  - Optimize your schema for most frequent use cases.
  - Do complex aggregation in the schema.

Example

- Suppose a client needs a database design for his blog/website.
- Website has the following requirements:
  - Every post has the unique title, description and url.
  - Every post can have one or more tags.
  - Every post has the name of its publisher and total number of likes.
  - Every post has comments given by users along with their name, message, data-time and likes.
  - On each post, there can be zero or more comments.

Example (2)

- In RDBMS:



- In MongoDB schema, design will have one collection:

```
{
        _id: POST_ID
        title: TITLE_OF_POST,
        description: POST_DESCRIPTION,
        by: POST_BY,
        url: URL_OF_POST,
        tags: [TAG1, TAG2, TAG3],
        likes: TOTAL_LIKES,
        comments: [
                { user:'COMMENT_BY',
                message: TEXT,
                dateCreated: DATE_TIME,
                like: LIKES },
                { user:'COMMENT_BY',
                message: TEXT,
                dateCreated: DATE_TIME,
                like: LIKES }
        ]
}
```

- While showing the data, in RDBMS you **need to join three tables** and in MongoDB, data will be shown from **one collection** only.

Create Database

- **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.
  - To check your currently selected database, use the command **db**
  - If you want to check your databases list, use the command **show dbs**.

```
> use mydb
switched to db mydb

> db
mydb

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

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

Drop Database

- **db.dropDatabase()** command is used to drop a existing database.
  - This will delete the selected database.

```
> use mydb
switched to db mydb

> db.mydb.insert( {“course”: “BD2”} )
WriteResult({ "nInserted" : 1 })

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

Create Collection

- **db.createCollection(name, options)** is used to create collection.

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

Field	Type	Description
capped	Boolean	(Optional) If true, enables a capped collection. Capped collection is a fixed size collection 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 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.

Example

- You can check the created collection by using the command **show collections**.

```
> use test
switched to db test

> db.createCollection("mycollection")
{ "ok" : 1 }

> show collections
test
mycollection

> db.createCollection("mycol", { capped : true, autoIndexId : true, size :
   6142800, max : 10000 } )
{ "ok" : 1 }
```

Example

- You don't need to create collection.
  - MongoDB creates collection automatically, when you insert some document.

```
> show collections
test

> db.course.insert( {"name" : ”BD2"} )
> show collections
course
test
```

- **db.collection.drop()** is used to drop a collection from the database.

```
> db.course.drop()
true
```

Datatypes

BSON is a binary serialization format
BSON extends JSON

- **String** − This is the most commonly used datatype to store the data.
- **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** − 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 binary data.
- **Code** − This datatype is used to store JavaScript code into the 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's **insert()** or **save()** method.

```
> db.post.insert( {
_id: ObjectId("7cd1b8673351579695abf75d"),
title: 'MongoDB Overview’,
description: 'MongoDB is no sql database’,
by: ‘BD2’,
url: 'http://www.unisa.it’,
tags: ['mongodb', 'database', 'NoSQL’],
likes: 100 })
```

Insert multiple documents

```
> db.post.insert([ {
        title: 'MongoDB Overview’,
        description: 'MongoDB is no sql database’,
        by: ‘BD2’,
        url: 'http://www.unisa.it’,
        tags: ['mongodb', 'database', 'NoSQL’],
        likes: 100 },
        { title: 'NoSQL Database’,
        description: "NoSQL database doesn't have tables",
        by: ‘BD2’,
        url: 'http://www.unisa.it’,
        tags: ['mongodb', 'database', 'NoSQL’],
        likes: 20,
        comments: [
                { user:'user1’,
                message: 'My first comment’,
                dateCreated: new Date(2020,11,10,2,35),
                like: 0 }
                ]
        }
])
```

Query Document

- **find()** method will display all the documents in a non-structured way.
  - To display the results in a formatted way, you can use **pretty()** method.

```
> db.post.find()
{ "_id" : ObjectId("5cd1b8673351579695abf75d"), "title" : "MongoDB
Overview", "description" : "MongoDB is no sql database", "by" : "BD2", "url" :
"http://www.unisa.it", "tags" : [ "mongodb", "database", "NoSQL" ], "likes" : 100 }
{ "_id" : ObjectId("5cd1b8673351579695abf75e"), "title" : "NoSQL Database",
"description" : "NoSQL database doesn't have tables", "by" : "BD2", "url" :
"http://www.unisa.it", "tags" : [ "mongodb", "database", "NoSQL" ], "likes" : 20,
"comments" : [ { "user" : "user1", "message" : "My first comment",
"dateCreated" : ISODate("2018-12-10T01:35:00Z"), "like" : 0 } ] }

> db.post.find().pretty()
…
```

Conditions

- To query the document on the basis of some condition, you can use following operations.

Operation	Syntax	Example	RDBMS Equivalent
Equality	**{<key>:<value>}**	db.post.find({"by":"BD2"}).pretty()	where by = 'BD2'
Less Than	**{<key>:{$lt:<value>}}**	db.post.find({"likes":{$lt:50}}).pretty()	where likes < 50
Less Than Equals	**{<key>:{$lte:<value>}}**	db.post.find({"likes":{$lte:50}}).pretty()	where likes <= 50
Greater Than	**{<key>:{$gt:<value>}}**	db.post.find({"likes":{$gt:50}}).pretty()	where likes > 50
Greater Than Equals	**{<key>:{$gte:<value>}}**	db.post.find({"likes":{$gte:50}}).pretty()	where likes >= 50
Not Equals	**{<key>:{$ne:<value>}}**	db.post.find({"likes":{$ne:50}}).pretty()	where likes != 50

AND/OR in MongoDB

- In the **find()** method, if you pass multiple keys by separating them by ',' then MongoDB treats it as **AND** condition.

```
> db.post.find({"likes": {$gt:10}, "by": "BD2"})
…

> db.post.find({$and:[{"by":"BD2"},{"title": "MongoDB Overview"}]})
…

> db.post.find({$or:[{"by":"BD2"},{"title": "MongoDB Overview"}]})
…

> db.post.find({"likes": {$gt:10}, $or: [{"by": "BD2"},{"title": "MongoDB Overview"}]})
…

> db.post.find({"likes": {$gt:10, $lt:200}, $or: [{"by": "BD2"},{"title": "MongoDB Overview"}]})
…
```

Update Document

- **update()** and **save()** methods are used to update document into a collection.
  - The update() method updates the values in the existing document;
  - the save() method replaces the existing document with the document passed in save() method.

- By default only a single document is updated.
  - To update multiple documents, set a parameter 'multi' to true.

```
> db.post.update({title: 'MongoDB Overview'},{$set:{title: 'New MongoDB Tutorial'}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })

> db.post.update({by: 'BD2'},{$set:{by: 'Basi Dati 2'}},{multi:true})
WriteResult({ "nMatched" : 5, "nUpserted" : 0, "nModified" : 5 })

> db.post.save({_id: ObjectId("5cd1b86733515795695abf75e"), title: "Css", by: "TSW"})
 …
```

Delete Document

- **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.
  - Without deletion criteria, then whole documents from the collection are removed.

```
> db.post.remove({title: 'Css'})
WriteResult({ "nRemoved" : 1 })

> db.post.remove({})
…
```

Projection

- projection means selecting only the necessary data rather than selecting whole of the data of a document.
  - If a document has 5 fields and you need to show only 3, then select only 3 fields from them.
- **find()** method accepts second optional parameter that is list of fields that you want to retrieve.
  - **find()** method displays all fields of a document.
  - To limit this, you need to set a list of fields with value 1 (show) or 0 (hide).

```
> db.post.find({},{title:1, by:1})

> db.post.find({},{title:1, _id:0, by:1})
```

_id field is always displayed while executing **find()** method, if you don't want this field, then you need to set it as 0.

Limit Records

- To limit the records in MongoDB, you need to use **limit()** method.
  - The method accepts the number of documents that you want to be displayed.

- **skip()** accepts number type argument and is used to skip the number of documents.

```
> db.post.find({},{title:1, by:1}).limit(2)

> db.post.find({},{title:1, by:1}).limit(1).skip(2)
```

Sort Records

- **sort()** method sorts documents.
  - The method accepts a document containing a list of fields along with their sorting order.
  - To specify sorting order 1 (ascending order) and -1 (descending order) are used.

```
> db.post.find({},{title:1, likes: 1, _id:0}).sort({title:-1, likes: 1})
```

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 to process a large volume of data.
- 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.

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

> db.post.ensureIndex({title: 1, description: -1})
…
> db.post.getIndexes()
```

1 is for ascending order.
-1 for descending order.

Indexing options

- **ensureIndex()** method also accepts list of options (which are optional).

Parameter	Type	Description
background	Boolean	Builds the index in the background so that building an index does not block other database activities.
unique	Boolean	Creates a unique index so that the collection will not accept insertion of documents where the index key or keys match an existing value in the index.
dropDups	Boolean	Creates a unique index on a field that may have duplicates. MongoDB indexes only the first occurrence of a key and removes all documents from the collection that contain subsequent occurrences of that key.
sparse	Boolean	If true, the index only references documents with the specified field. These indexes use less space but behave differently in some situations (particularly sorts).
weights	document	The weight is a number ranging from 1 to 99,999 and denotes the significance of the field relative to the other indexed fields in terms of the score.
default_language	string	For a text index, the language that determines the list of stop words and the rules for the stemmer and tokenizer. The default value is **english**.

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.
- For the aggregation you should use **aggregate()** method.

```
> db.post.aggregate([{$group : {_id : "$description", count : {$sum : 1}}}])
{ "_id" : "NoSQL database doesn't have tables", "count" : 2 }
{ "_id" : "MongoDB is no sql database", "count" : 4 }

> db.post.aggregate([{$group : {_id : "$description", likes : {$sum : "$likes"}}}])
{ "_id" : "NoSQL database doesn't have tables", "likes" : 40 }
{ "_id" : "MongoDB is no sql database", "likes" : 264 }
```

Aggregation expressions

Expression	Description	Example
$sum	Sums up the defined value from all documents in the collection.	db.post.aggregate([{$group : {_id : "$by", likes : {$sum : "$likes"}}}])
$avg	Calculates the average of all given values from all documents in the collection.	db.post.aggregate([{$group : {_id : "$by", likes : {$avg : "$likes"}}}])
$min	Gets the minimum of the corresponding values from all documents in the collection.	db.post.aggregate([{$group : {_id : "$by", likes : {$min : "$likes"}}}])
$max	Gets the maximum of the corresponding values from all documents in the collection.	db.post.aggregate([{$group : {_id : "$by", likes : {$max : "$likes"}}}])
$push	Inserts the value to an array in the resulting document.	db.post.aggregate([{$group : {_id : "$by", url : {$push: "$url"}}}])
$addToSet	Inserts the value to an array in the resulting document but does not create duplicates.	db.post.aggregate([{$group : {_id : "$by", url : {$addToSet : "$url"}}}])
$first	Gets the first document from the source documents according to the grouping.	db.post.aggregate([{$group : {_id : "$by", first_url : {$first : "$url"}}}])
$last	Gets the last document from the source documents according to the grouping.	db.post.aggregate([{$group : {_id : "$by", last_url : {$last : "$url"}}}])

Conversion during the data loading

- You can easily convert the string data type to numerical data type.
- You have two options: $toInt or $convert

- $toInt: **{ $toInt: <expr> }**
  - Converts a value to an integer.
    - If the value cannot be converted to an integer, $toInt returns errors.
    - If the value is null or missing, $toInt returns null.
- The $toInt is a shorthand for the following $convert expression: **{ $convert: { input: <expr>, to: "int" } }**

Example

- Create a collection **orders** with the following documents:

```
> db.orders.insert( [
   { _id: 1, item: "apple", qty: 5, price: 10 },
   { _id: 2, item: "pie", qty: 10, price: NumberDecimal("20.0") },
   { _id: 3, item: "ice cream", qty: "2", price: "4.99" },
   { _id: 4, item: "almonds" ,  qty: 5, price: 5 }
] )
```

- The following aggregation operation on the orders collection converts the **qty** to an integer as well as convert **price** to a decimal before calculating the total price.

Example

```
// Define stage to add convertedPrice and convertedQty fields with the converted price and qty values
> priceQtyConversionStage = {
   $addFields: {
      convertedPrice: { $toDecimal : "$price" },
      convertedQty: { $toInt : "$qty" },
   }
};

// Define stage to calculate total price by multiplying convertedPrice and convertedQty fields
> totalPriceCalculationStage = {
   $project: { item: 1, totalPrice: { $multiply: [ "$convertedPrice", "$convertedQty" ] }}
};


> db.orders.aggregate( [
   priceQtyConversionStage,
   totalPriceCalculationStage
])
```

Example

- The operation returns the following documents:

```
{ "_id" : 1, "item" : "apple", "totalPrice" : NumberDecimal("50.0000000000000") }
{ "_id" : 2, "item" : "pie", "totalPrice" : NumberDecimal("200.0") }
{ "_id" : 3, "item" : "ice cream", "totalPrice" : NumberDecimal("9.98") }
{ "_id" : 4, "item" : "almonds", "totalPrice" : NumberDecimal("25.00000000000000") }
```

- If the conversion operation encounters an error, the aggregation operation stops and throws an error.
  - To override this behavior, use $convert instead.

```
> priceQtyConversionStage = {
   $addFields: {
      convertedPrice: { $toDecimal : "$price" },
      convertedQty: { $convert : { input: "$qty", to: "int" } },
   }
};
```

Replication

- Replication is the process of synchronizing data across multiple servers.
- Replication provides redundancy and increases data availability with multiple copies of data on different database servers.
- Replication protects a database from the loss of a single server.
- Replication also allows you to recover from hardware failure and service interruptions.

Why Replication?

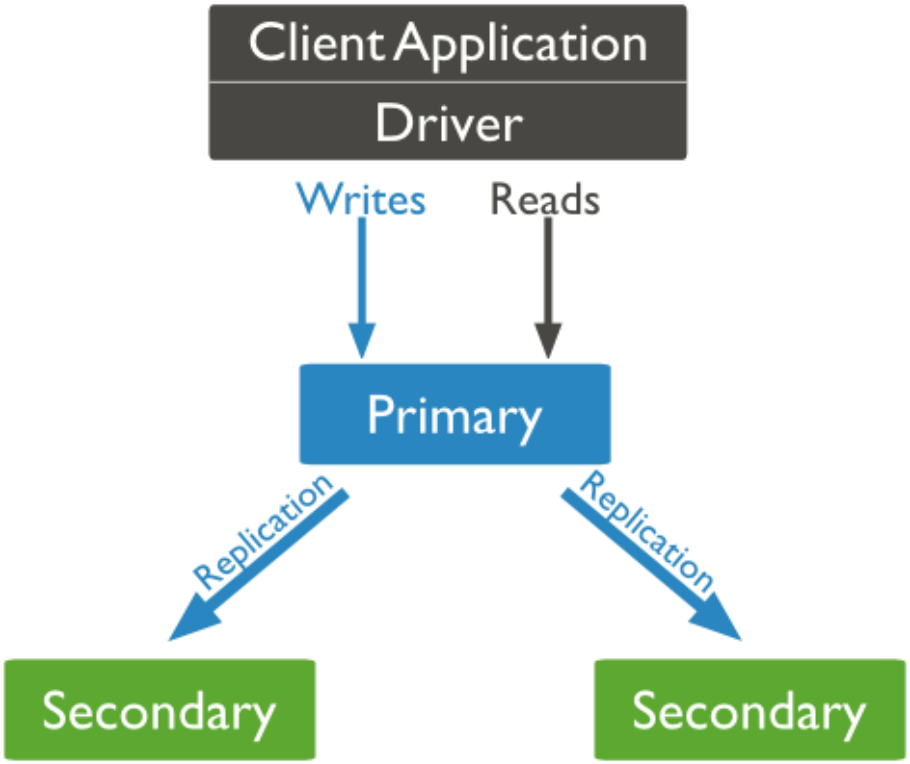
- To keep your data safe
- High (24*7) availability of data
- Disaster recovery
- No downtime for maintenance (like backups, index rebuilds, compaction)
- Read scaling (extra copies to read from)
- Replica set is transparent to the application

How Replication Works

- MongoDB achieves replication by the use of **replica set**.
- A replica set is a group of **mongod** instances that host the same data set.
- Replica set is a group of two or more nodes (generally minimum 3 nodes are required).
- In a replica, one node is primary node that receives all write operations.
  - All other instances, such as secondaries, apply operations from the primary so that they have the same data set.
  - Replica set can have only one primary node.
  - Remaining nodes are secondary.
  - All data replicates from primary to secondary node.
  - At the time of automatic failover or maintenance, election establishes for primary and a new primary node is elected.
  - After the recovery of failed node, it again join the replica set and works as a secondary node.

Replication diagram

- Client application always interacts with the primary node and the primary node then replicates the data to the secondary nodes.



Replica Set Features

- A cluster of N nodes
- Any one node can be primary
- All write operations go to primary
- Automatic failover
- Automatic recovery
- Consensus election of primary

Set Up a Replica Set

- Shutdown already running MongoDB server.
- Start the MongoDB server by specifying **-- replSet** option.

```
mongod --port "PORT" --dbpath "YOUR_DB_DATA_PATH" --replSet
"REPLICA_SET_INSTANCE_NAME"

mongod --port 27017 --dbpath "data/db" --replSet rs0
```

- It will start a **mongod** instance with the name **rs0**, on port **27017**.
- Now start the command prompt and connect to this mongod instance.
- The command **rs.initiate()** initiates a new replica set.
- To check the replica set configuration, issue the command **rs.conf()**.
- To check the status of replica run the command **rs.status()**.

Add Members to Replica Set

- To add members to replica set, start mongod instances on multiple machines.
- Now start a mongo client and issue a command **rs.add()**.

```
rs.add(HOST_NAME:PORT)

> rs.add("mongod1.net:27017")
```

- You can add mongod instance to replica set only when you are connected to primary node.
- To check whether you are connected to primary or not, run the command **db.isMaster()**.

Sharding

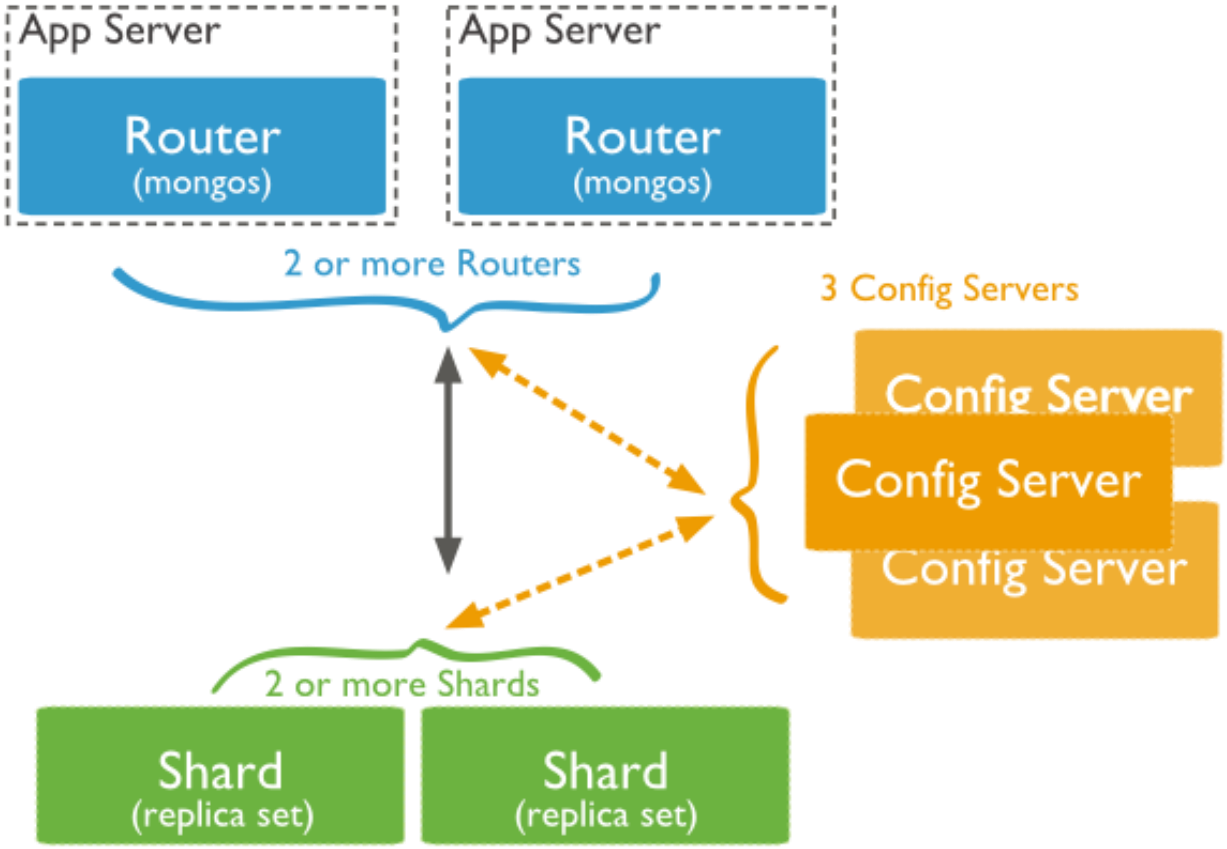
- Sharding is the process of storing data records across multiple machines and it is MongoDB's approach to meeting the demands of data growth.
- As the size of the data increases, a single machine may not be sufficient to store the data nor provide an acceptable read and write throughput.
- Sharding solves the problem with horizontal scaling.
- With sharding, you add more machines to support data growth and the demands of read and write operations.

Why Sharding?

- In replication, all writes go to master node
- Latency sensitive queries still go to master
- Single replica set has limitation of 12 nodes
- Memory can't be large enough when active dataset is big
- Local disk is not big enough
- Vertical scaling is too expensive

Sharding (2)

- The following diagram shows the sharding in MongoDB using sharded cluster.

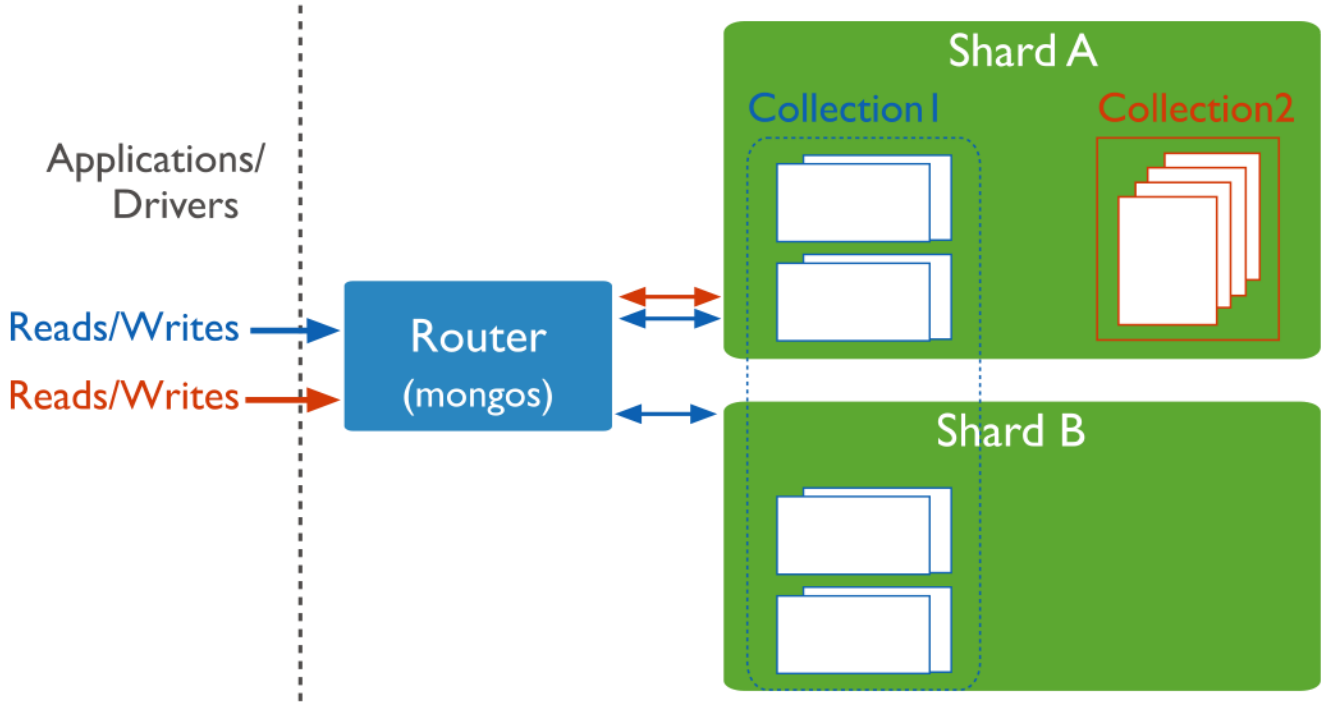


Sharding (3)

- **Shards** − Shards are used to store data. They provide high availability and data consistency.
  - Each shard is a separate replica set.
- **Config Servers** − Config servers store the cluster's metadata.
  - This data contains a mapping of the cluster's data set to the shards.
  - The query router uses this metadata to target operations to specific shards.
- **Query Routers** − Query routers are basically mongo instances, interface with client applications and direct operations to the appropriate shard.
  - The query router processes and targets the operations to shards and then returns results to the clients.
  - A sharded cluster can contain more than one query router to divide the client request load. A client sends requests to one query router.

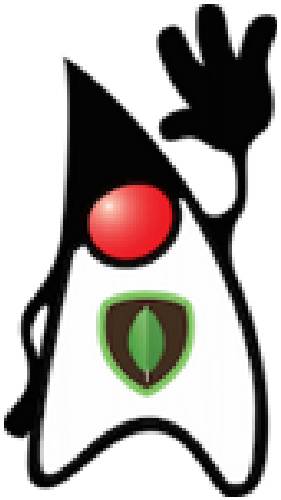
Connecting to a Sharded Cluster

- A database can have a mixture of sharded and unsharded collections.
  - Sharded collections are partitioned and distributed across the shards in the cluster.



MongoDB - Java

- You need to make sure that you have MongoDB JDBC driver and Java set up on the machine.
  - https://mongodb.github.io/mongo-java-driver/



Connect to Database

```
import com.mongodb.client.MongoDatabase;
import com.mongodb.MongoClient;
import com.mongodb.MongoCredential;

public class ConnectToDB {

   public static void main( String args[] ) {

      // Creating a Mongo client
      MongoClient mongo = new MongoClient( "localhost" , 27017 );

      // Creating Credentials
      MongoCredential credential;
      credential = MongoCredential.createCredential("sampleUser", "myDb",
         "password".toCharArray());
      System.out.println("Connected to the database successfully");

      // Accessing the database
      MongoDatabase database = mongo.getDatabase("myDb");
      System.out.println("Credentials ::"+ credential);
   }
}
```

```
> db.createUser( { user: "sampleUser",
                   pwd: "password",
                   roles: [    { role: "readWrite", db: "myDb" },
                               { role: "read", db: "reporting" }
                   ] } )
```

Create a Collection

```
import com.mongodb.client.MongoDatabase;
import com.mongodb.MongoClient;
import com.mongodb.MongoCredential;

public class CreatingCollection {

   public static void main( String args[] ) {

      // Creating a Mongo client
      MongoClient mongo = new MongoClient( "localhost" , 27017 );

      // Creating Credentials
      MongoCredential credential;
      credential = MongoCredential.createCredential("sampleUser", "myDb",
         "password".toCharArray());
      System.out.println("Connected to the database successfully");

      //Accessing the database
      MongoDatabase database = mongo.getDatabase("myDb");

      //Creating a collection
      database.createCollection("sampleCollection");
      System.out.println("Collection created successfully");
   }
}
```

Getting/Selecting a Collection

```
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;

import org.bson.Document;
import com.mongodb.MongoClient;
import com.mongodb.MongoCredential;

public class selectingCollection {

   public static void main( String args[] ) {

      // Creating a Mongo client
      MongoClient mongo = new MongoClient( "localhost" , 27017 );

      // Creating Credentials
      MongoCredential credential;
      credential = MongoCredential.createCredential("sampleUser", "myDb",
         "password".toCharArray());
      System.out.println("Connected to the database successfully");

      // Accessing the database
      MongoDatabase database = mongo.getDatabase("myDb");

      // Retieving a collection
      MongoCollection<Document> collection = database.getCollection("sampleCollection");
      System.out.println("Collection sampleCollection selected successfully");
   }
}
```

Insert a Document

```
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;

import org.bson.Document;
import com.mongodb.MongoClient;
import com.mongodb.MongoCredential;

public class InsertingDocument {

   public static void main( String args[] ) {

      // Creating a Mongo client
      MongoClient mongo = new MongoClient( "localhost" , 27017 );

      // Creating Credentials
      MongoCredential credential;
      credential = MongoCredential.createCredential("sampleUser", "myDb",
         "password".toCharArray());
      System.out.println("Connected to the database successfully");

      // Accessing the database
      MongoDatabase database = mongo.getDatabase("myDb");

      // Retrieving a collection
      MongoCollection<Document> collection = database.getCollection("sampleCollection");
      System.out.println("Collection sampleCollection selected successfully");

      Document document = new Document("title", "MongoDB")
      .append("id", 1)
      .append("description", "database")
      .append("likes", 100)
      .append("url", "http://www.unisa.it")
      .append("by", "BD2");
      collection.insertOne(document);
      System.out.println("Document inserted successfully");
   }
}
```

Retrieve All Documents

```
import com.mongodb.client.FindIterable;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;

import java.util.Iterator;
import org.bson.Document;
import com.mongodb.MongoClient;
import com.mongodb.MongoCredential;

public class RetrievingAllDocuments {

   public static void main( String args[] ) {

      // Creating a Mongo client
      MongoClient mongo = new MongoClient( "localhost" , 27017 );

      // Creating Credentials
      MongoCredential credential;
      credential = MongoCredential.createCredential("sampleUser", "myDb",
         "password".toCharArray());
      System.out.println("Connected to the database successfully");

      // Accessing the database
      MongoDatabase database = mongo.getDatabase("myDb");

      // Retrieving a collection
      MongoCollection<Document> collection = database.getCollection("sampleCollection");
      System.out.println("Collection sampleCollection selected successfully");

      // Getting the iterable object
      FindIterable<Document> iterDoc = collection.find();
      int i = 1;

      // Getting the iterator
      Iterator it = iterDoc.iterator();

      while (it.hasNext()) {
         System.out.println(it.next());
      i++;
      }
   }
}
```

Update Document

```
import com.mongodb.client.FindIterable;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import com.mongodb.client.model.Filters;
import com.mongodb.client.model.Updates;

import java.util.Iterator;
import org.bson.Document;
import com.mongodb.MongoClient;
import com.mongodb.MongoCredential;

public class UpdatingDocuments {

   public static void main( String args[] ) {

      // Creating a Mongo client
      MongoClient mongo = new MongoClient( "localhost" , 27017 );

      // Creating Credentials
      MongoCredential credential;
      credential = MongoCredential.createCredential("sampleUser", "myDb",
         "password".toCharArray());
      System.out.println("Connected to the database successfully");

      // Accessing the database
      MongoDatabase database = mongo.getDatabase("myDb");

      // Retrieving a collection
      MongoCollection<Document> collection = database.getCollection("sampleCollection");
      System.out.println("Collection myCollection selected successfully");

      collection.updateOne(Filters.eq("id", 1), Updates.set("likes", 150));
      System.out.println("Document update successfully...");

      // Retrieving the documents after updation
      // Getting the iterable object
      FindIterable<Document> iterDoc = collection.find();
      int i = 1;

      // Getting the iterator
      Iterator it = iterDoc.iterator();

      while (it.hasNext()) {
         System.out.println(it.next());
         i++;
      }
   }
}
```

Delete a Document

```
import com.mongodb.client.FindIterable;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import com.mongodb.client.model.Filters;

import java.util.Iterator;
import org.bson.Document;
import com.mongodb.MongoClient;
import com.mongodb.MongoCredential;

public class DeletingDocuments {

   public static void main( String args[] ) {

      // Creating a Mongo client
      MongoClient mongo = new MongoClient( "localhost" , 27017 );

      // Creating Credentials
      MongoCredential credential;
      credential = MongoCredential.createCredential("sampleUser", "myDb",
         "password".toCharArray());
      System.out.println("Connected to the database successfully");

      // Accessing the database
      MongoDatabase database = mongo.getDatabase("myDb");

      // Retrieving a collection
      MongoCollection<Document> collection = database.getCollection("sampleCollection");
      System.out.println("Collection sampleCollection selected successfully");

      // Deleting the documents
      collection.deleteOne(Filters.eq("id", 1));
      System.out.println("Document deleted successfully...");
   }
}
```

Relationships

- Relationships in MongoDB represent how various documents are logically related to each other.
- Relationships can be modeled via **Embedded** and **Referenced** approaches.
  - Such relationships can be either 1:1, 1:N, N:1 or M:N.
- *Let us consider the case of storing addresses for users.*
  - One user can have multiple addresses making this a 1:N relationship.

Example

- **users** document

```
{ "_id":ObjectId("52ffc33cd85242f436000001"),
"name": "Tom Hanks",
"contact": "987654321",
"dob": "01-01-1991"
}
```

- **address** document

```
{ "_id":ObjectId("52ffc4a5d85242602e000000"),
"building": "22 A, Indiana Apt",
"pincode": 123456,
"city": "Los Angeles",
"state": "California"
}
```

Modeling Embedded Relationships

- In the embedded approach, we will embed the address document inside the user document.

```
{ "_id":ObjectId("52ffc33cd85242f436000001"),
"contact": "987654321",
"dob": "01-01-1991",
"name": "Tom Benzamin",
"address": [
            { "building": "22 A, Indiana Apt",
            "pincode": 123456,
            "city": "Los Angeles",
            "state": "California" },
            { "building": "170 A, Acropolis Apt",
            "pincode": 456789,
            "city": "Chicago",
            "state": "Illinois" }
            ]
}
```

Modeling Embedded Relationships (2)

- This approach maintains all the related data in a single document, which makes it easy to retrieve and maintain.
- The whole document can be retrieved in a single query:

```
> db.users.findOne({"name":"Tom Benzamin"},{"address":1})
```

- The drawback is that if the embedded document keeps on growing too much in size, it can impact the read/write performance.

Modeling Referenced Relationships

- This is the approach of designing normalized relationship.
  - In this approach, both the user and address documents will be maintained separately but the user document will contain a field that will reference the address document's **id** field.

```
{ "_id":ObjectId("52ffc33cd85242f436000001"),
"contact": "987654321",
"dob": "01-01-1991",
"name": "Tom Benzamin",
"address_ids": [
            ObjectId("52ffc4a5d85242602e000000"),
            ObjectId("52ffc4a5d85242602e000001")
            ]
}
```

Modeling Referenced Relationships (2)

- With this approach, we will need two queries:
  - first to fetch the **address_ids** fields from **user** document and
  - second to fetch these addresses from **address** collection.

```
> var result = db.users.findOne({"name":"Tom Benzamin"},{"address_ids":1})

> db.address.find({"_id":{"$in":result["address_ids"]}})
```

- To print a variable:
  - **print(result)**
  - **printjson(result)**

Covered Queries

- What is a Covered Query?
  - All the fields in the query are part of an index.
  - All the fields returned in the query are in the same index.

- Since all the fields present in the query are part of an index, MongoDB matches the query conditions and returns the result using the same index without actually looking inside the documents.
  - Since indexes are present in RAM, fetching data from indexes is much faster as compared to fetching data by scanning documents.

Covered Queries (2)

- Create a compound index for the **users** collection on the fields **gender** and **user_name**
  - MongoDB would not go looking into database documents.

```
> db.users.ensureIndex({gender:1,user_name:1})
```

- It would fetch the required data from indexed data which is very fast.

```
> db.users.find({gender:"M"},{user_name:1,_id:0})
```

- The following query would not have been covered inside the index created above.

```
> db.users.find({gender:"M"},{email:1})
```

Analyzing Queries

- Analyzing queries is a very important aspect of measuring how effective the database and indexing design is:
  - **$explain**
  - **$hint**

- The **$explain** operator provides information on the query, indexes used in a query and other statistics.
- The **$hint** operator forces the query optimizer to use the specified index to run a query.

```
> db.users.find({gender:"M"},{user_name:1,_id:0}).explain()
…

> db.users.find({gender:"M"},{user_name:1,_id:0}).hint({gender:1,user_name:1})
…

> db.users.find({gender:"M"},{user_name:1,_id:0}).hint({gender:1,user_name:1}).explain()
…
```

Map Reduce

- **Map-reduce** is a data processing paradigm for condensing large volumes of data into useful aggregated results.
  - **mapReduce** command for map-reduce operations.

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

- **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

```
> db.posts.insert({
            "post_text": "BD2 is an amazing course",
            "user_name": "mark",
            "status":"active" })
… //insert 4 posts of users mark and lenz
```

- Use a mapReduce function on our **posts** collection to select all the active posts, group them on the basis of **user_name** and then count the number of posts by each user

```
> db.posts.mapReduce(
            function() { emit(this.user_name,1); },
            function(key, values) {return Array.sum(values)}, {
            query:{status:"active"}, out:"post_total" } )
{
            "result" : "post_total",
            "timeMillis" : 88,
            "counts" : {
                        "input" : 4, "emit" : 4,
                        "reduce" : 2, "output" : 2
            },
            "ok" : 1
}
```

Using MapReduce (2)

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

```
> db.posts.mapReduce(
            function() { emit(this.user_name,1); },
            function(key, values) {return Array.sum(values)}, {
            query:{status:"active"},
            out:"post_total" } ).find()
{ "_id" : "lenz", "value" : 2 }
{ "_id" : "mark", "value" : 2 }
```

Text Search

- MongoDB started supporting text indexes to search inside string content.
- The **Text Search** uses stemming techniques to look for specified words in the string fields by dropping stemming stop words like **a, an, the,** etc.
  - Create a Text Index

```
> db.posts.ensureIndex({post_text:"text"})

> db.posts.find({$text:{$search:"BD2"}},{_id:0})
{"post_text" : "BD2 is an amazing course", "user_name" : "mark", "status" : "active" }

> db.posts.find({$text:{$search:"program"}},{_id:0})
{"post_text" : "Robot Programming is a fantastic course", "user_name" : "lenz",
"status" : "active" }

> db.posts.find({$text:{$search:"fant"}},{_id:0})
> // no output
```

Regular Expression

- MongoDB also provides functionality of regular expression for string pattern matching using the **$regex** operator.
  - MongoDB uses PCRE (Perl Compatible Regular Expression) as regular expression language.

```
> db.posts.find({post_text:{$regex:"BD2"}},{_id:0})
{"post_text" : "BD2 is an amazing course", "user_name" : "mark", "status" : "active" }

> db.posts.find({post_text:/BD2/},{_id:0})
{"post_text" : "BD2 is an amazing course", "user_name" : "mark", "status" : "active" }


> db.posts.find({post_text:{$regex:"bd",$options:"$i"}},{_id:0})
{ "post_text" : "BD2 is an amazing course", "user_name" : "mark", "status" : "active" }
{ "post_text" : "BD is a fantastic course", "user_name" : "lenz", "status" : "active" }

> db.posts.find({post_text:{$regex:"bd",$options:"$s"}},{_id:0})
> // no output
```

Geospatial Queries

- MongoDB supports query operations on geospatial data.
- To calculate geometry over an Earth-like sphere, store your location data as **GeoJSON objects**.
- To specify GeoJSON data, use an embedded document with:
  - a field named **type** that specifies the GeoJSON object type:
    - Point
    - LineString
    - Polygon
    - Multi…
  - a field named **coordinates** that specifies the object's coordinates (e.g., longitude and latitude).

Example

- Create a collection places with the following documents:

```
> db.places.insert( {
        name: "Central Park",
        location: { type: "Point", coordinates: [ -73.97, 40.77 ] },
        category: "Parks" } );

> db.places.insert( {
        name: "Sara D. Roosevelt Park",
        location: { type: "Point", coordinates: [ -73.9928, 40.7193 ] },
        category: "Parks" } );

> db.places.insert( {
        name: "Polo Grounds",
        location: { type: "Point", coordinates: [ -73.9375, 40.8303 ] },
        category: "Stadiums" } );
```

Example (2)

- The following operation creates a **2dsphere** index on the **location** field:

```
> db.places.createIndex( { location: "2dsphere" } )
```

- The following query uses the **$near** operator to return documents that are at least 1000 meters from and at most 5000 meters from the specified GeoJSON point, sorted in order from nearest to farthest:

```
> db.places.find( {
        location: {
                $near: { $geometry: {
                        type: "Point", coordinates: [ -73.9667, 40.78 ] },
                        $minDistance: 1000,
                        $maxDistance: 5000 } }
                }, {_id: 0} )
{"name" : "Central Park", "location" : { "type" : "Point", "coordinates" : [ -73.97, 40.77 ] },
"category" : "Parks" }
```

Example (3)

- The following operation uses the **$geoNear** aggregation operation to return documents that match the query filter { category: "Parks" }, sorted in order of nearest to farthest to the specified GeoJSON point:

```
> db.places.aggregate( [ {
        $geoNear: { near: { type: "Point", coordinates: [ -73.9667, 40.78 ] },
                    spherical: true,
                    query: { category: "Parks" },
                    distanceField: "calcDistance" }
        } ] )
{ "_id" : ObjectId("5cd9aeebc5e5354acae8502c"), "name" : "Central Park", "location" : {
"type" : "Point", "coordinates" : [ -73.97, 40.77 ] }, "category" : "Parks", "calcDistance" :
1147.4220523120696 }
{ "_id" : ObjectId("5cd9aeebc5e5354acae8502d"), "name" : "Sara D. Roosevelt Park",
"location" : { "type" : "Point", "coordinates" : [ -73.9928, 40.7193 ] }, "category" : "Parks",
"calcDistance" : 7106.506152782733 }
```