e-PGPathshala
Subject : Computer Science
Paper: Data Analytics
Module No 34: CS/DA/34 –MongoDB-III
Quadrant 1 – e-text

1.1 Introduction

This chapter gives an overview of the MongoDB's synchronization concept and database connectivity with Java and PHP.

1.2 Learning Outcome

- To understand the process of Replication in MongoDB
- To understand the need for horizontal scaling using sharding in MongoDB
- To learn MongoDB connectivity to other applications Development Frameworks

1.3 MongoDB - 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. With additional copies of the data, you can dedicate one to disaster recovery, reporting, or backup.

Why Replication?

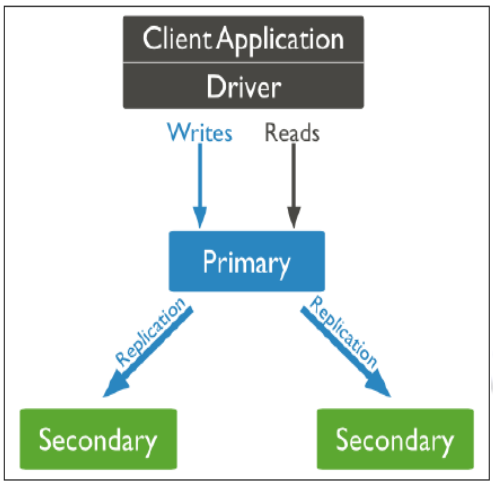
- To keep your data safe
- High (24*7) availability of data
- Disaster recovery
- No downtime for maintenance (like backups, index rebuilds, compaction)
- Read scaling (extra copies to read from)
- Replica set is transparent to the application

How Replication Works in MongoDB

- MongoDB achieves replication by the use of replica set. A replica set is a group of **mongod** instances that host the same data set. 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.
- Replica set is a group of two or more nodes (generally minimum 3 nodes are required).

- In a replica set, one node is primary node and 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 joins the replica set and works as a secondary node.

A typical diagram of MongoDB replication is shown in which client application always interact 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

In this tutorial, we will convert standalone MongoDB instance to a replica set. To convert to replica set, following are the steps:

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

Following is the basic syntax of --replSet:

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

Example

```
mongod --port 27017 --dbpath "D:\set up\mongodb\data" --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.
- In Mongo client, issue the command **rs.initiate()** to initiate a new replica set.
- To check the replica set configuration, issue the command **rs.conf()**. To check the status of replica set issue 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()**.

Syntax

The basic syntax of **rs.add()** command is as follows:

```
>rs.add (HOST_NAME: PORT)
```

Example

Suppose your mongod instance name is **mongod1.net** and it is running on port **27017**. To add this instance to replica set, issue the command **rs.add()** in Mongo client.

```
>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, issue the command **db.isMaster()** in Mongo client.

1.4 MongoDB - 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

There are two methods for addressing system growth: vertical and horizontal scaling.

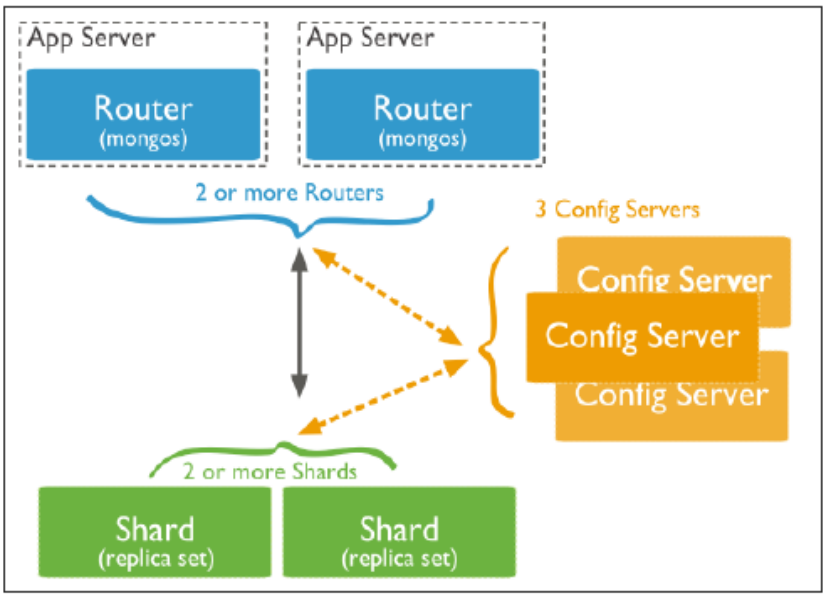
Vertical Scaling involves increasing the capacity of a single server, such as using a more powerful CPU, adding more RAM, or increasing the amount of storage space.

Horizontal Scaling involves dividing the system dataset and load over multiple servers, adding additional servers to increase capacity as required.

MongoDB supports *horizontal scaling* through sharding.

(i) **Sharding in MongoDB**

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



In the following diagram, there are three main components:

- **Shards**: Shards are used to store data. They provide high availability and data consistency. In production environment, 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. In production environment, sharded clusters have exactly 3 config servers.
- **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. Generally, sharded clusters have many query routers.

(ii) Shard Keys

To distribute the documents in a collection, MongoDB partitions the collection using the shard key. The shard key consists of an immutable field or fields that exist in every document in the target collection.

You choose the shard key when sharding a collection. The choice of shard key cannot be changed after sharding. A sharded collection can have only one shard key. See Shard Key Specification.To shard a non-empty collection, the collection must have an index that starts with the shard key. For empty collections, MongoDB creates the index if the collection does not already have an appropriate index for the specified shard key.

The choice of shard key affects the performance, efficiency, and scalability of a sharded cluster. A cluster with the best possible hardware and infrastructure can be bottlenecked by the choice of shard key. The choice of shard key and its backing index can also affect the sharding strategy that your cluster can use.

Chunks

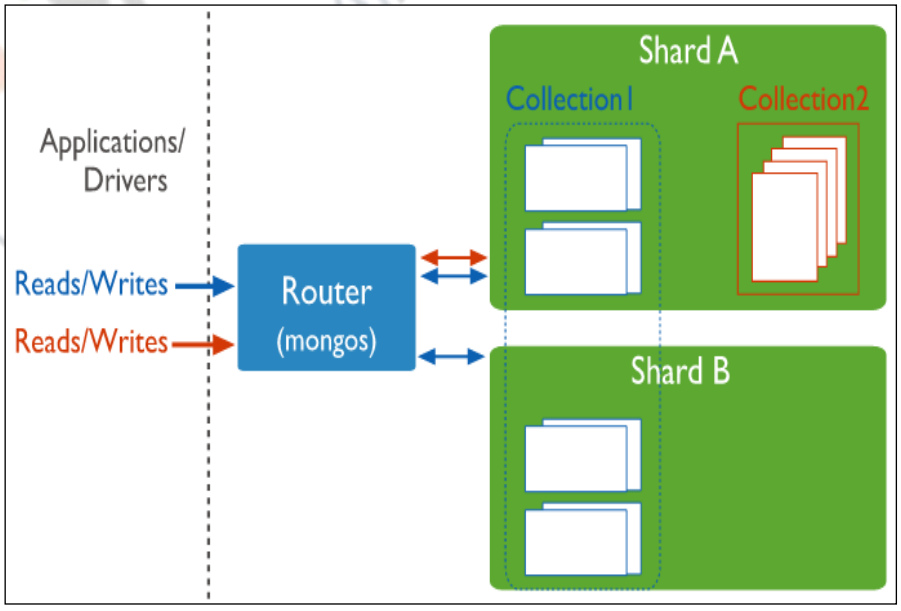
MongoDB partitions sharded data into chunks. Each chunk has an inclusive lower and exclusive upper range based on the shard key.MongoDB migrates chunks across the shards in the sharded cluster using the sharded cluster balancer. The balancer attempts to achieve an even balance of chunks across all shards in the cluster.

Sharded and Non-Sharded Collections

A database can have a mixture of sharded and unsharded collections. Sharded collections are partitioned and distributed across the shards in the cluster. Unsharded collections are stored on a primary shard. Each database has its own primary shard.

Connecting to a Sharded Cluster

You must connect to a mongos router to interact with any collection in the sharded cluster. This includes sharded and unsharded collections. Clients should never connect to a single shard in order to perform read or write operations.

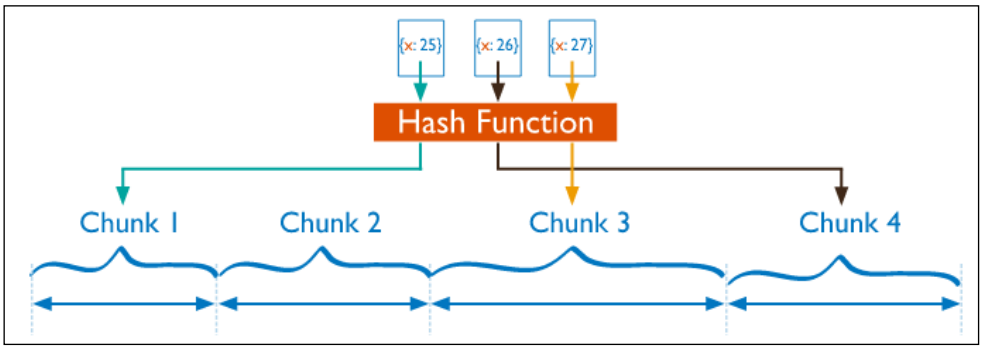


(iii) Sharding Strategy

MongoDB supports two sharding strategies for distributing data across sharded clusters.

Hashed Sharding

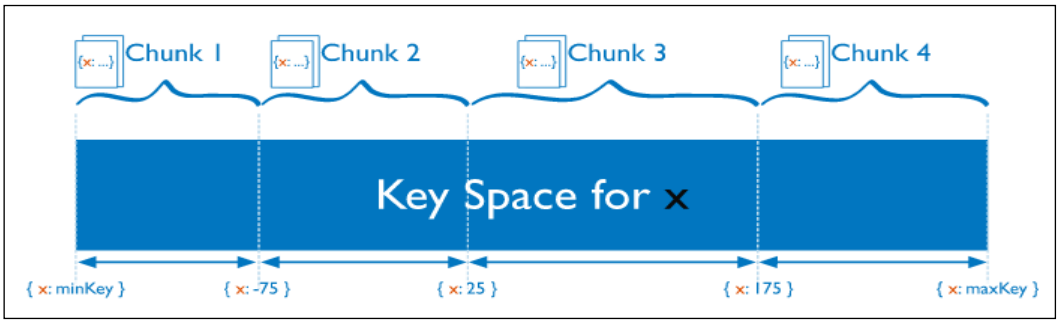
Hashed Sharding involves computing a hash of the shard key field's value. Each chunk is then assigned a range based on the hashed shard key values.



While a range of shard keys may be "close", their hashed values are unlikely to be on the same chunk. Data distribution based on hashed values facilitates more even data distribution, especially in data sets where the shard key changes monotonically.

Ranged Sharding

Ranged sharding involves dividing data into ranges based on the shard key values. Each chunk is then assigned a range based on the shard key values.



A range of shard keys whose values are "close" are more likely to reside on the same chunk. This allows for targeted operations as a mongos can route the operations to only the shards that contain the required data.

The efficiency of ranged sharding depends on the shard key chosen. Poorly considered shard keys can result in uneven distribution of data, which can negate some benefits of sharding or can cause performance bottlenecks. See shard key selection for ranged sharding. However, hashed distribution means that ranged-based queries on the shard key are less likely to target a single shard, resulting in more cluster wide broadcast operations

1.5 MongoDB - Create Backup and Deployment

(i) Dump MongoDB Data

To create backup of database in MongoDB, you should use **mongodump**command. This command will dump the entire data of your server into the dump directory.

There are many options available by which you can limit the amount of data or create backup of your remote server.

Syntax

The basic syntax of **mongodump** command is as follows: **>mongodump**

Example

Start your mongod server. Assuming that your mongod server is running on the localhost and port 27017, open a command prompt and go to the bin directory of your mongodb instance and type the command **mongodump**

Consider the mycol collection has the following data. **>mongodump.** Following is a list of available options that can be used with the **mongodump**command.

This command will backup only specified database at specified path.

Syntax	Description	Example
mongodump --host HOST_NAME --port PORT_NUMBER	This command will backup all databases of specified mongod instance	mongodump --host tutorialspoint.com --port 27017
mongodump --dbpath DB_PATH --out BACKUP_DIRECTORY		mongodump --dbpath /data/db/ --out /data/backup/
mongodump --collection COLLECTION --db DB_NAME	This command will backup only specified collection of specified database.	mongodump --collection mycol --db test

Restore Data

To restore backup data MongoDB's **mongorestore** command is used. This command restores all of the data from the backup directory.

Syntax

The basic syntax of **mongorestore** command is: **>mongorestore**

(ii) MongoDB - Deployment

When you are preparing a MongoDB deployment, you should try to understand how your application is going to hold up in production. It's a good idea to develop a consistent, repeatable approach to managing your deployment environment so that you can minimize any surprises once you're in production.

The best approach incorporates prototyping your setup, conducting load testing, monitoring key metrics, and using that information to scale your setup. The key part of the approach is to proactively monitor your entire system - this will help you understand how your production system will hold up before deploying, and determine where you will need to add capacity. Having insight into potential spikes in your memory usage, for example, could help put out a write-lock fire before it starts.

To monitor your deployment, MongoDB provides some of the following commands:

- ✓ **mongostat**

This command checks the status of all running mongod instances and return counters of database operations. These counters include inserts, queries, updates, deletes, and cursors. Command also shows when you're hitting page faults, and showcase your lock percentage. This means that you're running low on memory, hitting write capacity or have some performance issue.

To run the command, start your mongod instance. In another command prompt, go to **bin** directory of your mongodb installation and type **mongostat**.

D:\set up\mongodb\bin>mongostat

- ✓ **mongotop**

This command tracks and reports the read and writes activity of MongoDB instance on a collection basis. By default, **mongotop** returns information in each second, which you can change it accordingly. You should check that this read and write activity matches your application intention, and you're not firing too many writes to the database at a time, reading too frequently from a disk, or are exceeding your working set size.

To run the command, start your mongod instance. In another command prompt, go to **bin** directory of your mongodb installation and type **mongotop**.

```
D:\set up\mongodb\bin>mongotop
```

To change **mongotop** command to return information less frequently, specify a specific number after the mongotop command.

```
D:\set up\mongodb\bin>mongotop 30
```

The above example will return values every 30 seconds.

1.6 MongoDB – Java and PHP connectivity

(i) Installation

Before you start using MongoDB in your Java programs, you need to make sure that you have MongoDB JDBC driver and Java set up on the machine. You can check Java tutorial for Java installation on your machine. Now, let us check how to set up MongoDB JDBC driver.

- You need to download the jar from the path Download mongo.jar. Make sure to download the latest release of it.
- You need to include the mongo.jar into your classpath.

Connect to Database

To connect database, you need to specify the database name, if the database doesn't exist then MongoDB creates it automatically.

```
try{
      // To connect to mongodb server
        MongoClient mongoClient = new MongoClient( "localhost" , 27017 );
```

```
      // Now connect to your databases
      DB db = mongoClient.getDB( "test" );
      System.out.println("Connect to database successfully");
      boolean auth = db.authenticate(myUserName, myPassword);
      System.out.println("Authentication: "+auth);

   }catch(Exception e){
      System.err.println( e.getClass().getName() + ": " + e.getMessage() );
   }
```

If you are going to use Windows machine, then you can compile and run your code as follows –

```
$javac MongoDBJDBC.java

$java -classpath ".; mongo-2.10.1.jar" MongoDBJDBC

Connect to database successfully

Authentication: true
```

(ii) MongoDB - PHP

To use MongoDB with PHP, you need to use MongoDB PHP driver. Download the driver from the url Download PHP Driver. Make sure to download the latest release of it. Now unzip the archive and put php_mongo.dll in your PHP extension directory ("ext" by default) and add the following line to your php.ini file −

extension = php_mongo.dll

Make a Connection and Select a Database

To make a connection, you need to specify the database name; if the database doesn't exist then MongoDB creates it automatically.

Following is the code snippet to connect to the database –

```
<?php
// connect to mongodb
$m = new MongoClient();
echo "Connection to database successfully";
// select a database
$db = $m->mydb;
echo "Database mydb selected";
?>
```

When the program is executed, it will produce the following result −

Connection to database successfully

Database mydb selected

1.7 Advanced concepts in MongoDB

Some of the advanced concepts of MongoDB such as relationship, atomic operations, advanced indexing, text search etc are discussed here.

(a) Relationship

Relationships in MongoDB represent how various documents are logically related to each other. Relationships can be modelled via **Embedded** and **Referenced** approaches. Such relationships can be either 1:1, 1: N, N: 1 or N: N. 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")
   ]
}
```

As shown above, the user document contains the array field **address_ids** which contains ObjectIds of corresponding addresses. Following queries fetch the **address_ids** fields from **user** document and fetch these addresses from **address** collection.

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

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

(b) Atomic Operations

MongoDB does not support **multi-document atomic transactions**. However, it does provide atomic operations on a single document. So if a document has hundred fields the update statement will either update all the fields or none, hence maintaining atomicity at the document-level.

Consider the following products document –

```
{
  "_id":1,
  "product_name": "Samsung S3",
  "category": "mobiles",
  "product_total": 5,
  "product_available": 3,
  "product_bought_by": [
    {
      "customer": "john",
      "date": "7-Jan-2014"
    },
    {
      "customer": "mark",
      "date": "8-Jan-2014"
    }
  ]
}
```

In this document, embed the information of the customer who buys the product in the **product_bought_by** field. Now, whenever a new customer buys the product, first check if the product is still available using **product_available** field. If available, reduce the value of product_available field as well as insert the new customer's embedded document in the product_bought_by field.

(c) Advanced Indexing

Consider the following document of the **user's** collection –

```
{
  "address": {
    "city": "Los Angeles",
    "state": "California",
    "pincode": "123"
  },
  "tags": [
    "music",
    "cricket",
    "blogs"
  ],
  "name": "Tom Benzamin"
}
```

The above document contains an **address sub-document** and a **tags array**.

- **Indexing Array Fields**

Creating an index on array in turn creates separate index entries for each of its fields. So create an index on tags array, separate indexes will be created for its values music, cricket and blogs.

To create an index on tags array, use the following code –

```
>db.users.ensureIndex ({"tags":1})
```

After creating the index, search on the tags field of the collection using find().

```
>db.users.find ({tags:"cricket"})
```

To verify that proper indexing is used, use the following **explain** command –

```
>db.users.find ({tags:"cricket"}).explain ()
```

The above command resulted in "cursor": "BtreeCursor tags_1" which confirms that proper indexing is used.

- **Indexing Sub-Document Fields**

To search documents based on city, state and pincode fields, create an index on all the fields of the sub-document. For creating an index on all the three fields of the sub-document, use the following code –

```
>db.users.ensureIndex ({"address.city":1,"address.state":1,"address.pincode":1})
```

Once the index is created, we can search for any of the sub-document fields utilizing this index as follows –

```
>db.users.find ({"address. city":"Los Angeles"})
```

Remember that the query expression has to follow the order of the index specified. So the index created above would support the following queries –

```
>db.users.find ({"address. city":"Los Angeles","address.state":"California"})
```

It will also support the following query –

```
>db.users.find ({"address.city":"LosAngeles","address.state":"California",
```

(d) Map Reduce

As per the MongoDB documentation, **Map-reduce** are a data processing paradigm for condensing large volumes of data into useful aggregated results. MongoDB uses **mapReduce** command for map-reduce operations. MapReduce is generally used for processing large data sets.

MapReduce Command

Following is the syntax of the basic mapReduce command –

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

The map-reduce function first queries the collection, then maps the result documents to emit key-value pairs, which is then reduced based on the keys that have multiple values.

In the above syntax −

- **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

(e) Regular Expressions

Regular Expressions are frequently used in all languages to search for a pattern or word in any string. 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. Unlike text search, no 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/})
```

(f) Text Search

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. At present, MongoDB supports around 15 languages.

- **Enabling Text Search**

Enable text search with the following code –

```
>db.adminCommand ({setParameter: true, textSearchEnabled: true})
```

- **Creating Text Index**

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

```
{
   "post_text": "enjoy the mongodb articles on tutorialspoint",
   "tags": [
      "mongodb",
      "tutorialspoint"
   ]
}
```

Create a text index on post_text field to search inside our posts' text –

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

- **Using Text Index**

Search for all the posts having the word **tutorialspoint** in their text.

```
>db.posts.find ({$text :{ $search:"tutorialspoint"}})
```

The above command returned the following result documents having the word **tutorialspoint** in their post text –

```
{
   "_id": ObjectId("53493d14d852429c10000002"),
   "post_text": "enjoy the mongodb articles on tutorialspoint",
   "tags": [ "mongodb", "tutorialspoint" ]
}
{
   "_id": ObjectId("53493d1fd852429c10000003"),
   "post_text": "writing tutorials on mongodb",
   "tags" : [ "mongodb", "tutorial" ]
}
```

Using Text Search highly improves the search efficiency as compared to normal search.

- **Deleting Text Index**

To delete an existing text index, first find the name of index using the following query –

```
>db.posts.getIndexes ()
```

After getting the name of your index from above query, run the following command. Here, **post_text_text** is the name of the index.

```
>db.posts.dropIndex("post_text_text")
```

SUMMARY

- Replication helps to recover from hardware failure and service interruptions
- Sharding, support data growth and the demands of read and write operations
- Replication and sharding are necessary for availability and partition in CAP