### Sharding Cluster – Hands On using Zips JSON Test Data

**Step 1: Import Sample Data**

First, you need to import the zips.json data into your MongoDB cluster. You can download the zips.json file from here.

1. **Import Data**:

mongoimport --host HP830G5-EliteBook --port 26000 --db test --collection zips --file zips.json

**Step 2: Enable Sharding for the Database**

Once the data is imported, you need to enable sharding on the test database.

1. **Enable Sharding on the Database**:

sh.enableSharding("test")

**Step 3: Shard the Collection**

Next, shard the zips collection on a sharding key. A good choice for sharding key in this dataset might be { state: 1, city: 1 }.

1. **Shard the Collection**:

sh.shardCollection("test.zips", { state: 1, city: 1 })

**Step 4: Verify Sharding**

You can verify that sharding is working by checking the distribution of the data across the shards.

1. **Check the Sharding Status**:

sh.status()

1. **Check Chunk Distribution**:

db.printShardingStatus()

**Step 5: Query the Data**

Perform some queries to ensure that data is being correctly distributed and accessed.

1. **Sample Query**:

db.zips.findOne()

1. **Query for a Specific State and City**:

db.zips.find({ state: "NY", city: "ALBANY" })

**Example of the Steps in mongosh**

// Connect to the mongos instance

mongosh --host HP830G5-EliteBook --port 26000

// Switch to the 'test' database

use test

// Enable sharding on the 'test' database

sh.enableSharding("test")

// Shard the 'zips' collection on { state: 1, city: 1 }

sh.shardCollection("test.zips", { state: 1, city: 1 })

// Check the sharding status

sh.status()

// Check chunk distribution

db.printShardingStatus()

// Perform a sample query

db.zips.findOne()

// Query for a specific state and city

db.zips.find({ state: "NY", city: "ALBANY" })

**Summary**

1. Import the zips.json data into your MongoDB cluster.
2. Enable sharding on the test database.
3. Shard the zips collection using an appropriate sharding key.
4. Verify the sharding status and chunk distribution.
5. Perform queries to ensure data distribution and access.

Note - Show me using either find or aggregation query on the zips database and indexing and query execution statistics before we apply Sharding key based search

**Step 2: Perform Queries**

Let's perform some queries on the zips collection.

1. **Find Query**:

// Find a document in the zips collection

db.zips.findOne({ state: "NY", city: "ALBANY" })

1. **Aggregation Query**:

// Aggregate the total population by state

db.zips.aggregate([

{ $group: { \_id: "$state", totalPopulation: { $sum: "$pop" } } },

{ $sort: { totalPopulation: -1 } }

])

**Step 3: Create Indexes**

Let's create indexes to optimize query performance.

1. **Create Indexes**:

// Create an index on state and city

db.zips.createIndex({ state: 1, city: 1 })

// Create an index on population

db.zips.createIndex({ pop: 1 })

**Step 4: Gather Query Statistics**

MongoDB provides various ways to gather statistics about query performance.

1. **Explain Plan for Find Query**:

// Get the explain plan for a find query

db.zips.find({ state: "NY", city: "ALBANY" }).explain("executionStats")

1. **Explain Plan for Aggregation Query**:

// Get the explain plan for an aggregation query

db.zips.aggregate([

{ $group: { \_id: "$state", totalPopulation: { $sum: "$pop" } } },

{ $sort: { totalPopulation: -1 } }

]).explain("executionStats")

**Example Commands in mongosh**

// Connect to the mongos instance

mongosh --host HP830G5-EliteBook --port 26000

// Switch to the 'test' database

use test

// Perform a find query

db.zips.findOne({ state: "NY", city: "ALBANY" })

// Perform an aggregation query

db.zips.aggregate([

{ $group: { \_id: "$state", totalPopulation: { $sum: "$pop" } } },

{ $sort: { totalPopulation: -1 } }

])

// Create indexes

db.zips.createIndex({ state: 1, city: 1 })

db.zips.createIndex({ pop: 1 })

// Get the explain plan for a find query

db.zips.find({ state: "NY", city: "ALBANY" }).explain("executionStats")

// Get the explain plan for an aggregation query

db.zips.aggregate([

{ $group: { \_id: "$state", totalPopulation: { $sum: "$pop" } } },

{ $sort: { totalPopulation: -1 } }

]).explain("executionStats")

**Step 5: Analysing Results**

Review the output of the explain("executionStats") to understand the query execution performance. The explain output includes important information such as:

* **Execution Time**: Time taken to execute the query.
* **Number of Documents Scanned**: Number of documents scanned during the query execution.
* **Indexes Used**: Indexes used by the query.

**Summary**

1. Perform basic queries and aggregation on the zips collection.
2. Create indexes to optimize query performance.
3. Use the explain function to gather query execution statistics.

Let us discuss everything and details about the .explain() method, how to check query execution details and what each parameter output is about when we fire "executionStats()

The .explain() method in MongoDB is a powerful tool that provides detailed information about how MongoDB executes a query. This information is essential for understanding the performance of your queries and optimizing them.

**Overview of .explain()**

When you call .explain() on a query, MongoDB returns a document that describes the execution plan of the query. You can specify different verbosity levels to get more or less detail:

* "queryPlanner": Provides information about the query plan.
* "executionStats": Provides detailed information about the query execution, including performance metrics.
* "allPlansExecution": Provides detailed information about the execution of all plans considered by the query optimizer.

**Usage**

You can use .explain() with various MongoDB operations such as find, aggregate, update, and delete.

**Example Usage**

// Find query with explain

db.zips.find({ state: "NY", city: "ALBANY" }).explain("executionStats")

// Aggregate query with explain

db.zips.aggregate([

{ $group: { \_id: "$state", totalPopulation: { $sum: "$pop" } } },

{ $sort: { totalPopulation: -1 } }

]).explain("executionStats")

**Output of .explain("executionStats")**

When you specify "executionStats", the output document includes several important sections:

1. **queryPlanner**: Information about the query plan.
2. **executionStats**: Detailed execution statistics.
3. **serverInfo**: Information about the server that executed the query.

Let's break down these sections in detail.

**1. queryPlanner**

The queryPlanner section provides information about how MongoDB plans to execute the query.

* **plannerVersion**: Version of the query planner used.
* **namespace**: The namespace (database and collection) on which the query is executed.
* **indexFilterSet**: Whether index filters were used.
* **parsedQuery**: The query as parsed by MongoDB.
* **winningPlan**: The query plan selected by the query optimizer.
* **rejectedPlans**: Plans considered but not selected by the optimizer.

**Example**

{

"queryPlanner": {

"plannerVersion": 1,

"namespace": "test.zips",

"indexFilterSet": false,

"parsedQuery": {

"state": "NY",

"city": "ALBANY"

},

"winningPlan": {

"stage": "FETCH",

"inputStage": {

"stage": "IXSCAN",

"keyPattern": {

"state": 1,

"city": 1

},

"indexName": "state\_1\_city\_1",

"direction": "forward"

}

},

"rejectedPlans": []

}

}

**2. executionStats**

The executionStats section provides detailed metrics about the execution of the query.

* **nReturned**: Number of documents returned by the query.
* **executionTimeMillis**: Total time in milliseconds to execute the query.
* **totalKeysExamined**: Number of index keys examined.
* **totalDocsExamined**: Number of documents examined.
* **executionStages**: Details about each stage of the execution plan.

**Example**

{

"executionStats": {

"executionSuccess": true,

"nReturned": 1,

"executionTimeMillis": 2,

"totalKeysExamined": 1,

"totalDocsExamined": 1,

"executionStages": {

"stage": "FETCH",

"nReturned": 1,

"executionTimeMillisEstimate": 1,

"works": 2,

"advanced": 1,

"needTime": 0,

"needYield": 0,

"saveState": 0,

"restoreState": 0,

"isEOF": 1,

"docsExamined": 1,

"alreadyHasObj": 0,

"inputStage": {

"stage": "IXSCAN",

"nReturned": 1,

"executionTimeMillisEstimate": 0,

"works": 2,

"advanced": 1,

"needTime": 0,

"needYield": 0,

"saveState": 0,

"restoreState": 0,

"isEOF": 1,

"keyPattern": {

"state": 1,

"city": 1

},

"indexName": "state\_1\_city\_1",

"isMultiKey": false,

"multiKeyPaths": {

"state": [],

"city": []

},

"keysExamined": 1,

"dupsTested": 0,

"dupsDropped": 0,

"seenInvalidated": 0,

"direction": "forward",

"indexBounds": {

"state": ["[\"NY\", \"NY\"]"],

"city": ["[\"ALBANY\", \"ALBANY\"]"]

}

}

}

}

}

**3. serverInfo**

The serverInfo section provides information about the server where the query was executed.

* **host**: The hostname of the server.
* **port**: The port number.
* **version**: The MongoDB server version.
* **gitVersion**: The Git version of the MongoDB server.

**Example**

{

"serverInfo": {

"host": "HP830G5-EliteBook",

"port": 26000,

"version": "5.0.6",

"gitVersion": "0c84a925d95a6767a93f522a4b15aafea0a4b5e1"

}

}

**Analysing the Output**

1. **Execution Time**: Look at executionTimeMillis to understand how long the query took to execute.
2. **Documents and Keys Examined**: Compare totalKeysExamined and totalDocsExamined with nReturned to see if indexes are effectively reducing the number of documents MongoDB must examine.
3. **Winning Plan**: Check the winningPlan to see which indexes were used.
4. **Rejected Plans**: Examine rejectedPlans to understand other plans the optimizer considered but did not choose.

**MongoDB Query Execution Statistics**

MongoDB provides detailed information about how queries are executed through the explain() method. This method can be used to retrieve the query execution plan, which includes both estimated and actual execution statistics.

**1. Estimated Query Execution**

The explain() method can be used with the "queryPlanner" verbosity to get an estimated execution plan. This plan provides insights into how MongoDB plans to execute the query without actually running it.

**Example:**

db.employees.find({ EmployeeID: 1 }).explain("queryPlanner")

**Output:**

{

"queryPlanner": {

"plannerVersion": 1,

"namespace": "test.employees",

"indexFilterSet": false,

"parsedQuery": { "EmployeeID": { "$eq": 1 } },

"winningPlan": {

"stage": "COLLSCAN",

"direction": "forward"

},

"rejectedPlans": []

}

}

**2. Actual Query Execution**

To obtain the actual execution statistics, use the "executionStats" verbosity. This will execute the query and provide detailed statistics on the execution process.

**Example:**

db.employees.find({ EmployeeID: 1 }).explain("executionStats")

**Output:**

{

"queryPlanner": { /\* ... same as above ... \*/ },

"executionStats": {

"executionSuccess": true,

"nReturned": 1,

"executionTimeMillis": 1,

"totalKeysExamined": 0,

"totalDocsExamined": 1000,

"executionStages": {

"stage": "COLLSCAN",

"nReturned": 1,

"executionTimeMillisEstimate": 0,

"works": 1002,

"advanced": 1,

"needTime": 1000,

"needFetch": 0,

"saveState": 7,

"restoreState": 7,

"isEOF": 1,

"docsExamined": 1000

}

}

}

**3. Detailed Attributes in Execution Statistics**

**Query Planner**

* **plannerVersion**: Version of the query planner used.
* **namespace**: The namespace (database and collection) the query is executed on.
* **indexFilterSet**: Whether index filters are set for the query.
* **parsedQuery**: The parsed query in MongoDB's internal format.
* **winningPlan**: The plan selected by the query planner as the best way to execute the query.
* **rejectedPlans**: Alternative plans that were considered but not chosen.

**Execution Stats**

* **executionSuccess**: Indicates if the query execution was successful.
* **nReturned**: Number of documents returned by the query.
* **executionTimeMillis**: Time taken to execute the query in milliseconds.
* **totalKeysExamined**: Total number of index keys examined.
* **totalDocsExamined**: Total number of documents examined.
* **executionStages**: Detailed breakdown of each stage in the query execution pipeline.

**4. Execution Stages**

Execution stages represent the steps taken to execute the query. Each stage has specific attributes, such as:

* **stage**: The type of operation (e.g., COLLSCAN for collection scan).
* **nReturned**: Number of documents returned by this stage.
* **executionTimeMillisEstimate**: Estimated time for the stage execution.
* **works**: Number of times the stage had to work (process data).
* **advanced**: Number of times the stage produced a result.
* **needTime**: Number of times the stage needed more time to produce a result.
* **needFetch**: Number of times the stage needed to fetch data.
* **saveState**: Number of times the stage had to save its state (e.g., for yielding).
* **restoreState**: Number of times the stage had to restore its state.
* **isEOF**: Indicates if the stage has reached the end of the data.
* **docsExamined**: Number of documents examined by this stage.

**Example: Analyzing a Query**

Consider a query that retrieves employees with a specific EmployeeID:

db.employees.find({ EmployeeID: 1 }).explain("allPlansExecution")

**Output:**

{

"queryPlanner": { /\* ... \*/ },

"executionStats": {

"executionSuccess": true,

"nReturned": 1,

"executionTimeMillis": 1,

"totalKeysExamined": 0,

"totalDocsExamined": 1000,

"executionStages": {

"stage": "COLLSCAN",

"nReturned": 1,

"executionTimeMillisEstimate": 0,

"works": 1002,

"advanced": 1,

"needTime": 1000,

"needFetch": 0,

"saveState": 7,

"restoreState": 7,

"isEOF": 1,

"docsExamined": 1000

}

},

"allPlansExecution": [

{

"stage": "COLLSCAN",

"filter": { "EmployeeID": { "$eq": 1 } },

"nReturned": 1,

"executionTimeMillis": 1,

"totalDocsExamined": 1000

}

]

}

**Summary**

* **Estimated Execution Plan**: Provides a theoretical plan and costs without running the query. Useful for understanding how MongoDB plans to execute the query.
* **Actual Execution Plan**: Provides detailed execution statistics, including actual time taken, documents examined, and execution stages. Useful for performance tuning and identifying bottlenecks.
* **Execution Stages**: Breaks down the execution process into stages, showing detailed metrics for each stage.

Using these insights, you can optimize queries by, for example, creating indexes, adjusting query structure, or modifying database schema to improve performance.

Setting up replication and sharding in MongoDB Atlas involves creating a cluster and configuring it for high availability and scalability.

Below are step-by-step guidelines to perform replication and sharding on MongoDB Atlas.

**1. Setting Up a MongoDB Atlas Account**

1. **Sign Up/Log In**:
   * Go to [MongoDB Atlas](https://www.mongodb.com/cloud/atlas).
   * Sign up for a new account or log in if you already have one.

**2. Creating a Cluster with Replication**

1. **Create a New Project**:
   * Click on the "Projects" menu and create a new project.
   * Name your project and click "Next".
2. **Build a New Cluster**:
   * Click on the "Build a Cluster" button.
   * Choose a cloud provider (e.g., AWS, Google Cloud, Azure).
   * Select your preferred region.
   * Choose a cluster tier (e.g., M0 (Free Tier) for testing, or M10 and above for production).
3. **Cluster Configuration**:
   * In the cluster configuration, MongoDB Atlas sets up replication by default. For production environments, ensure you choose a cluster tier that supports multiple replicas (e.g., M10+).
4. **Additional Settings**:
   * Configure additional settings as needed (e.g., backup, advanced configuration).
5. **Create Cluster**:
   * Click "Create Cluster" to initiate the cluster creation process. This may take a few minutes.

**3. Configuring Network Access**

1. **Whitelist IP Address**:
   * Navigate to the "Network Access" section.
   * Click "Add IP Address" and add the IP addresses that are allowed to connect to your MongoDB Atlas cluster. You can use 0.0.0.0/0 to allow access from anywhere (not recommended for production).
2. **Create a Database User**:
   * Go to the "Database Access" section.
   * Click "Add New Database User".
   * Create a user with appropriate roles (e.g., readWrite, dbAdmin).

**4. Connecting to the Cluster**

1. **Get Connection String**:
   * In the "Clusters" section, click "Connect" next to your cluster.
   * Choose your connection method (e.g., MongoDB Shell, application, Compass).
   * Copy the connection string and replace <username>, <password>, and <dbname> with your actual credentials.

**5. Enabling Sharding**

1. **Access Cluster Configuration**:
   * In the Atlas UI, click on your cluster name to go to the cluster configuration page.
2. **Sharding Configuration**:
   * Click on "Collections" under your cluster.
   * Click on "Enable Sharding" for the database you want to shard.
3. **Add Shard Key**:
   * Choose the collection you want to shard.
   * Click on the collection and then click "Enable Sharding".
   * Select a shard key. A shard key is a field or combination of fields that MongoDB uses to distribute documents among shards. Choose a shard key that ensures even distribution of data.
4. **Monitor Sharding**:
   * MongoDB Atlas provides monitoring tools to view the status of sharded clusters. You can see how data is distributed across shards and adjust your shard key if necessary.

**Summary**

* **Replication**: MongoDB Atlas handles replication automatically when you create a cluster with a tier that supports it. This ensures high availability and data redundancy.
* **Sharding**: Sharding is configured on a per-database basis. Choose a shard key that ensures even distribution and enables sharding through the Atlas UI.

**Example: Connecting to Your Sharded Cluster**

Here’s an example of connecting to your sharded MongoDB Atlas cluster using MongoDB Shell:

shell

Copy code

mongo "mongodb+srv://<cluster-name>.mongodb.net/<dbname>" --username <username> --password <password>

Replace <cluster-name>, <dbname>, <username>, and <password> with your actual cluster name, database name, username, and password.

**How to set up a MongoDB sharded cluster with three config servers, three shard servers, and two routers on a single CentOS 9 system using MongoDB Enterprise Edition 7.0.**

**Prerequisites**

1. **Install MongoDB Enterprise Edition 7.0**: Follow the official MongoDB [installation guide](https://docs.mongodb.com/manual/tutorial/install-mongodb-on-red-hat/) to install MongoDB Enterprise Edition 7.0 on CentOS 9.

**Step-by-Step Setup**

**1. Start Config Servers**

Config servers store metadata and configuration settings for the cluster.

1. **Create Config Server Directories**:

mkdir -p /data/config1 /data/config2 /data/config3

1. **Start Config Server Instances**:

mongod --configsvr --replSet configReplSet --dbpath /data/config1 --port 27019 --bind\_ip localhost --fork --logpath /var/log/mongodb/config1.log

mongod --configsvr --replSet configReplSet --dbpath /data/config2 --port 27020 --bind\_ip localhost --fork --logpath /var/log/mongodb/config2.log

mongod --configsvr --replSet configReplSet --dbpath /data/config3 --port 27021 --bind\_ip localhost --fork --logpath /var/log/mongodb/config3.log

1. **Initialize Config Server Replica Set**: Connect to one of the config servers and initialize the replica set.

mongo --port 27019

rs.initiate({

\_id: "configReplSet",

configsvr: true,

members: [

{ \_id: 0, host: "localhost:27019" },

{ \_id: 1, host: "localhost:27020" },

{ \_id: 2, host: "localhost:27021" }

]

})

**2. Start Shard Servers**

Each shard is a replica set.

1. **Create Shard Directories**:

mkdir -p /data/shard1 /data/shard2 /data/shard3

1. **Start Shard Instances**:

mongod --shardsvr --replSet shard1ReplSet --dbpath /data/shard1 --port 27022 --bind\_ip localhost --fork --logpath /var/log/mongodb/shard1.log

mongod --shardsvr --replSet shard2ReplSet --dbpath /data/shard2 --port 27023 --bind\_ip localhost --fork --logpath /var/log/mongodb/shard2.log

mongod --shardsvr --replSet shard3ReplSet --dbpath /data/shard3 --port 27024 --bind\_ip localhost --fork --logpath /var/log/mongodb/shard3.log

1. **Initialize Shard Replica Sets**:
   * For shard1:

mongo --port 27022

rs.initiate({

\_id: "shard1ReplSet",

members: [

{ \_id: 0, host: "localhost:27022" }

]

})

* + For shard2:

mongo --port 27023

rs.initiate({

\_id: "shard2ReplSet",

members: [

{ \_id: 0, host: "localhost:27023" }

]

})

* + For shard3:

mongo --port 27024

rs.initiate({

\_id: "shard3ReplSet",

members: [

{ \_id: 0, host: "localhost:27024" }

]

})

**3. Start Routers (mongos)**

The routers direct queries to the appropriate shards.

1. **Start mongos Instances**:

mongos --configdb configReplSet/localhost:27019,localhost:27020,localhost:27021 --port 27025 --bind\_ip localhost --fork --logpath /var/log/mongodb/mongos1.log

mongos --configdb configReplSet/localhost:27019,localhost:27020,localhost:27021 --port 27026 --bind\_ip localhost --fork --logpath /var/log/mongodb/mongos2.log

**4. Add Shards to the Cluster**

Connect to one of the mongos instances and add the shards to the cluster.

1. **Add Shards**:

mongo --port 27025

sh.addShard("shard1ReplSet/localhost:27022")

sh.addShard("shard2ReplSet/localhost:27023")

sh.addShard("shard3ReplSet/localhost:27024")

**5. Verify the Cluster**

1. **Check Shard Status**:

mongo --port 27025

sh.status()

The output should show the three shards and the configuration servers.

**Summary**

You now have a MongoDB sharded cluster set up on a single CentOS 9 system, including:

* 3 config servers, configured as a replica set.
* 3 shards, each configured as a replica set.
* 2 mongos routers.

Below is an automated shell script that sets up three config servers, three shard servers, and two routers on a single CentOS 9 system. This script assumes you have MongoDB Enterprise Edition installed and that you have sudo privileges. The script handles directory creation, configuration file creation, and starting the MongoDB instances.

**Automated Shell Script**

#!/bin/bash

# Define MongoDB base path and ports

MONGO\_BASE\_PATH="/data"

LOG\_BASE\_PATH="/var/log/mongodb"

CONFIG\_PORTS=(27019 27020 27021)

SHARD\_PORTS=(27022 27023 27024)

MONGOS\_PORTS=(27025 27026)

# Ensure MongoDB directories exist

sudo mkdir -p $MONGO\_BASE\_PATH/config{1,2,3}

sudo mkdir -p $MONGO\_BASE\_PATH/shard{1,2,3}

sudo mkdir -p $LOG\_BASE\_PATH

# Create and configure config servers

for i in {1..3}; do

CONFIG\_PATH="$MONGO\_BASE\_PATH/config$i"

PORT=${CONFIG\_PORTS[$i-1]}

LOG\_PATH="$LOG\_BASE\_PATH/config$i.log"

CONFIG\_FILE="/etc/mongod\_config$i.conf"

sudo tee $CONFIG\_FILE <<EOF

systemLog:

destination: file

path: $LOG\_PATH

logAppend: true

storage:

dbPath: $CONFIG\_PATH

net:

bindIp: localhost

port: $PORT

replication:

replSetName: configReplSet

sharding:

clusterRole: configsvr

EOF

sudo mongod --config $CONFIG\_FILE --fork

# Initialize the config server replica set

sleep 5

mongo --port ${CONFIG\_PORTS[0]} <<EOF

rs.initiate({

\_id: "configReplSet",

configsvr: true,

members: [

{ \_id: 0, host: "localhost:${CONFIG\_PORTS[0]}" },

{ \_id: 1, host: "localhost:${CONFIG\_PORTS[1]}" },

{ \_id: 2, host: "localhost:${CONFIG\_PORTS[2]}" }

]

})

EOF

# Create and configure shard servers

for i in {1..3}; do

SHARD\_PATH="$MONGO\_BASE\_PATH/shard$i"

PORT=${SHARD\_PORTS[$i-1]}

LOG\_PATH="$LOG\_BASE\_PATH/shard$i.log"

CONFIG\_FILE="/etc/mongod\_shard$i.conf"

sudo tee $CONFIG\_FILE <<EOF

systemLog:

destination: file

path: $LOG\_PATH

logAppend: true

storage:

dbPath: $SHARD\_PATH

net:

bindIp: localhost

port: $PORT

replication:

replSetName: shard${i}ReplSet

sharding:

clusterRole: shardsvr

EOF

sudo mongod --config $CONFIG\_FILE --fork

# Initialize the shard replica sets

for i in {1..3}; do

PORT=${SHARD\_PORTS[$i-1]}

mongo --port $PORT <<EOF

rs.initiate({

\_id: "shard${i}ReplSet",

members: [

{ \_id: 0, host: "localhost:$PORT" }

]

})

EOF

# Create and configure routers (mongos)

for i in {1..2}; do

PORT=${MONGOS\_PORTS[$i-1]}

LOG\_PATH="$LOG\_BASE\_PATH/mongos$i.log"

CONFIG\_FILE="/etc/mongos$i.conf"

sudo tee $CONFIG\_FILE <<EOF

systemLog:

destination: file

path: $LOG\_PATH

logAppend: true

net:

bindIp: localhost

port: $PORT

sharding:

configDB: configReplSet/localhost:${CONFIG\_PORTS[0]},localhost:${CONFIG\_PORTS[1]},localhost:${CONFIG\_PORTS[2]}

EOF

sudo mongos --config $CONFIG\_FILE --fork

# Add shards to the cluster

sleep 10

mongo --port ${MONGOS\_PORTS[0]} <<EOF

sh.addShard("shard1ReplSet/localhost:${SHARD\_PORTS[0]}")

sh.addShard("shard2ReplSet/localhost:${SHARD\_PORTS[1]}")

sh.addShard("shard3ReplSet/localhost:${SHARD\_PORTS[2]}")

EOF

echo "MongoDB sharded cluster setup completed successfully."

**Instructions**

1. **Save the Script**: Save the above script to a file, for example, setup\_mongodb\_cluster.sh.
2. **Make the Script Executable**:

chmod +x setup\_mongodb\_cluster.sh

1. **Run the Script**:

sudo ./setup\_mongodb\_cluster.sh

**Explanation**

* **Directories Creation**: Creates necessary directories for config servers and shard servers.
* **Configuration Files Creation**: Creates configuration files for each MongoDB instance (config servers, shard servers, and routers).
* **Starting MongoDB Instances**: Starts each MongoDB instance using the created configuration files.
* **Initializing Replica Sets**: Initializes the replica sets for config servers and shard servers.
* **Adding Shards**: Adds the shards to the cluster using the mongos instance.

This script automates the entire process of setting up a MongoDB sharded cluster on a single CentOS 9 system. Adjust the configuration paths, ports, and other settings as needed for your specific environment

Top of Form

Bottom of Form