Introduction

MongoDB is a **NoSQL** database that uses a **document-oriented** approach to simplify the management of large **datasets**. **Horizontal scaling** is one of MongoDB's essential features, enabled by the tool's native support for **database** sharding.

This article will show you how to deploy a sharded MongoDB cluster using Docker and Docker Compose.

Prerequisites

- Docker installed.
- Docker Compose installed.
- MongoDB Client **application**.

What Is Database Sharding?

Database sharding is a method of distributing large datasets across multiple servers called **shards**. Shards are often deployed as replica sets stored on multiple machines for **high availability**.

When a dataset is sharded, it is divided into smaller subsets based on criteria such as user ID, location, or date. Since each data subset resides on a separate server, the system determines which shard contains the relevant data and directs queries accordingly.

Sharding in MongoDB

Sharding leverages MongoDB's flexible document model to distribute data across shards. Additionally, allows MongoDB to support high throughput operations on large datasets.

By distributing data based on a shard key (e.g., a user ID or location), MongoDB ensures that queries are directed to the appropriate shard, maximizing throughput

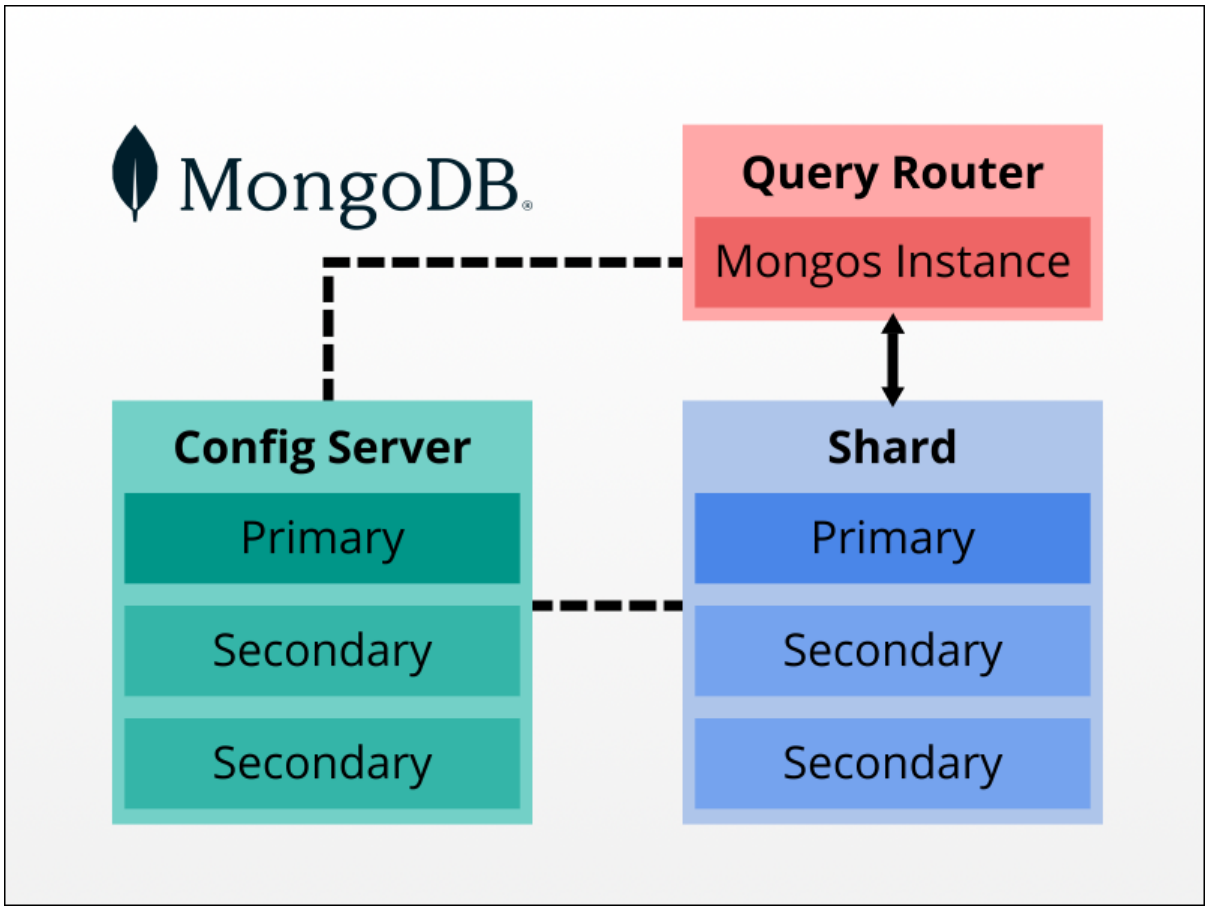
and minimizing **latency**. Features like automatic sharding and intelligent data balancing allow users to scale their apps seamlessly and efficiently.

Sharded Cluster Architecture and Components

Each sharded cluster in MongoDB consists of the following components:

- **Config servers**. Servers that store cluster configuration and **metadata**. One of the servers acts as the primary server, and others are secondary.
- **Shards**. Cluster members that contain data subsets. One shard is primary, while others are secondary.
- **Query router**. The system that enables client applications to interact with the cluster.

The diagram below illustrates the architecture of a sharded MongoDB cluster.



Benefits of Sharding in MongoDB

Sharding brings the following benefits to MongoDB users:

- **Improved querying speed.** The **database management system** needs to access only the relevant shard, limiting the amount of data it needs to process.
- **Easier horizontal scaling.** More servers can be added whenever necessary to handle growing data volumes and user traffic.
- **Better availability.** If one shard fails, the system continues to operate as other shards remain accessible.

When to Set up MongoDB Sharding

Consider setting up MongoDB Sharding in the following scenarios:

- **Rapidly growing data volume.** As a dataset expands, querying a single server can become inefficient and slow.
- **High write loads.** Sharding distributes write operations across multiple servers and prevents the bottleneck effect.
- **The need for high availability.** Other shards remain available if a cluster member fails.
- **Anticipated growth.** Sharding facilitates dealing with growing **data storage** needs and increased user traffic.
- **Unsatisfactory read performance.** Sharding significantly improves read performance, especially for geographically distributed applications.

Note: Sharding can increase **operational overhead**. There is also a potential for **data skew**, i.e., uneven data distribution across shards that can negatively impact performance.

How to Set Up Sharding in MongoDB

To deploy a fully functional MongoDB sharded cluster, **deploy each cluster element separately**. Below are the steps for sharded cluster deployment using **Docker containers** and Docker Compose.

> **Note**: The tutorial uses a single test machine to deploy all cluster elements. While it is possible to implement sharding in this way, MongoDB recommends using a separate machine for each member of each deployed replica set in a **production environment**.

Step 1: Deploy Config Server Replica Set

Start by deploying a replica set of config servers to store configuration settings and cluster metadata. Proceed with the steps below to create config servers:

1. **Create a directory using the mkdir command** and navigate to it using the **cd command**:

```
mkdir config && cd configCopy
```

2. Use a **text editor** to **create a file** called *compose.yml*:

```
nano compose.ymlCopy
```

3. Write the configuration you want to deploy. The example below defines three config server replicas in the **`services`** section and three **Docker volumes** for persistent data storage in the **`volumes`** section:

```
services:

  configs1:
    container_name: configs1
    image: mongo
    command: mongod --configsvr --replSet cfgrs --port 27017 --
dbpath /data/db
    ports:
      - 10001:27017
    volumes:
```

```
      - configs1:/data/db

  configs2:
    container_name: configs2
    image: mongo
    command: mongod --configsvr --replSet cfgrs --port 27017 --
dbpath /data/db
    ports:
      - 10002:27017
    volumes:
      - configs2:/data/db

  configs3:
    container_name: configs3
    image: mongo
    command: mongod --configsvr --replSet cfgrs --port 27017 --
dbpath /data/db
    ports:
      - 10003:27017
    volumes:
      - configs3:/data/db

volumes:
  configs1: {}
  configs2: {}
  configs3: {}Copy
```

Each config server replica requires the following parameters:

- **Name**. Choose any name you want. Numbered names are recommended for easier instance management.
- **Name of the Docker container**. Choose any name.
- **Docker image**. Use the **`mongo`** image available on Docker Hub.
- **`mongod` command**. The command specifies the instance is a config server (**`--configsvr`**) and part of a replica set (**`--replSet`**). Furthermore, it defines the default **port** (**`27017`**) and the path to the database.
- **Ports**. Map the default Docker port to an external port of your choosing.
- **Volumes**. Define the database path on a permanent storage volume.

Note: Learn about the **differences between Docker images and containers**.

4. Save and exit the file when you finish.

5. Apply the configuration with the **`docker compose`** command:

```
docker compose up -dCopy
```

The system confirms the successful deployment of the MongoDB config servers.

```
marko@phoenixnap:~/config$ docker compose up -d
[+] Running 12/12
 ✔ configs2 Pulled                                                    27.6s
 ✔ configs3 Pulled                                                    27.6s
   ✔ 726b8a513d66 Pull complete                                        7.1s
   ✔ 50d8ee45ae7b Pull complete                                        7.1s
   ✔ 238f502c2b61 Pull complete                                        7.6s
   ✔ e3740760faa6 Pull complete                                        8.2s
   ✔ 8b9a1b1cb415 Pull complete                                        8.2s
   ✔ 1998aad8b060 Pull complete                                        8.3s
   ✔ 86287f367904 Pull complete                                        8.3s
   ✔ 63675dc18c9f Pull complete                                       24.0s
   ✔ 6d97d47a654a Pull complete                                       24.0s
 ✔ configs1 Pulled                                                    27.6s
[+] Running 7/7
 ✔ Network config_default     Created                                  0.1s
 ✔ Volume "config_configs1"   Created                                  0.0s
 ✔ Volume "config_configs2"   Created                                  0.0s
 ✔ Volume "config_configs3"   Created                                  0.0s
 ✔ Container configs1         Started                                  2.0s
 ✔ Container configs2         Started                                  2.1s
 ✔ Container configs3         Started                                  2.1s
marko@phoenixnap:~/config$
```

6. Use the **`docker compose`** command to list only the containers relevant to the deployment:

```
docker compose psCopy
```

All three config server replicas show as separate containers with different external ports.

```
marko@phoenixnap:~/config$ docker compose ps
NAME       IMAGE          COMMAND                  SERVICE    CREATED
STATUS          PORTS
configs1   mongo:4.2.22   "docker-entrypoint.s…"   configs1   4 minutes ago
Up 4 minutes   0.0.0.0:10001->27017/tcp, :::10001->27017/tcp
configs2   mongo:4.2.22   "docker-entrypoint.s…"   configs2   4 minutes ago
Up 4 minutes   0.0.0.0:10002->27017/tcp, :::10002->27017/tcp
configs3   mongo:4.2.22   "docker-entrypoint.s…"   configs3   4 minutes ago
Up 4 minutes   0.0.0.0:10003->27017/tcp, :::10003->27017/tcp
marko@phoenixnap:~/config$
```

7. Check the Docker volumes:

```
docker volume lsCopy
```

```
marko@phoenixnap:~/config$ docker volume ls
DRIVER    VOLUME NAME
local     9b6b0c0ec1006bf3c8183d641e849f80597c00b0f099287b200c9d8407ecdff5
local     be0873706af53676b30d6f66e9467ce71e684ec05866144aa9207454a1af6f55
local     c43f3bf88ebca576587d08ce1e34132461d02a7d77d957240d25ffdb23b55bd1
local     config_configs1
local     config_configs2
local     config_configs3
marko@phoenixnap:~/config$
```

8. Use the Mongo client application to log in to one of the config server replicas:

```
mongosh mongodb://[ip_address]:[port]Copy
```

As a result, the MongoDB shell command prompt appears:

```
marko@phoenixnap:~/config$ mongosh mongodb://192.168.0.25:10001
Current Mongosh Log ID: 6756b2fdaca91bb7ece94969
Connecting to:          mongodb://192.168.0.25:10001/?directConnection=true&ap
pName=mongosh+2.3.4
Using MongoDB:          4.2.22
Using Mongosh:          2.3.4

For mongosh info see: https://www.mongodb.com/docs/mongodb-shell/

test>
```

9. Initiate the replicas in MongoDB by using the **`rs.initiate()`** method. The **`configsvr`** field set to **`true`** is required for config server initiation:

```
rs.initiate(
  {
    _id: "cfgrs",
    configsvr: true,
    members: [
      { _id : 0, host : "[ip_address]:[port]" },
      { _id : 1, host : "[ip_address]:[port]" },
      { _id : 2, host : "[ip_address]:[port]" }
    ]
  }
)Copy
```

If the operation is successful, the **`"ok"`** value in the output is **`1`**. Conversely, if an error occurs, the value is **`0`**, and an error message is displayed.

```
test> rs.initiate(
...   {
...     _id: "cfgrs",
...     configsvr: true,
...     members: [
...       { _id : 0, host : "192.168.0.25:10001" },
...       { _id : 1, host : "192.168.0.25:10002" },
...       { _id : 2, host : "192.168.0.25:10003" },
...     ]
...    }
... )
{
  ok: 1,
  '$gleStats': {
    lastOpTime: Timestamp({ t: 1733735654, i: 1 }),
    electionId: ObjectId('000000000000000000000000')
  },
  lastCommittedOpTime: Timestamp({ t: 0, i: 0 })
}
test>
```

Press **Enter** to exit the secondary and return to the primary instance.

10. Use the **`rs.status()`** method to check the status of your instances:

`rs.status()`Copy

```
members: [
  {
    _id: 0,
    name: '192.168.0.25:10001',
    health: 1,
    state: 1,
    stateStr: 'PRIMARY',
    uptime: 3525,
    optime: { ts: Timestamp({ t: 1733736009, i: 1 }), t: Long('1') },
    optimeDate: ISODate('2024-12-09T09:20:09.000Z'),
    syncingTo: '',
    syncSourceHost: '',
    syncSourceId: -1,
    infoMessage: '',
    electionTime: Timestamp({ t: 1733735665, i: 1 }),
    electionDate: ISODate('2024-12-09T09:14:25.000Z'),
    configVersion: 1,
    self: true,
    lastHeartbeatMessage: ''
  },
```

11. Exit the MongoDB shell by entering:

`.exit`Copy

Step 2: Create Shard Replica Sets

After setting up a config server replica set, create data shards. The example below shows how to create and initiate a single shard replica set, but the process for each subsequent shard is the same:

1. Create and navigate to the **directory** where you will store shard-related manifests:

```
mkdir shard && cd shardCopy
```

2. Create a *compose.yml* file with a text editor:

```
nano compose.ymlCopy
```

3. Configure shard instances. Below is an example of a *compose.yml* that defines three shard replica sets and three permanent storage volumes:

```
services:

  shard1s1:
    container_name: shard1s1
    image: mongo
    command: mongod --shardsvr --replSet shard1rs --port 27017 --
dbpath /data/db
    ports:
      - 20001:27017
    volumes:
      - shard1s1:/data/db

  shard1s2:
    container_name: shard1s2
    image: mongo
    command: mongod --shardsvr --replSet shard1rs --port 27017 --
dbpath /data/db
    ports:
      - 20002:27017
    volumes:
      - shard1s2:/data/db

  shard1s3:
    container_name: shard1s3
    image: mongo
```

```
    command: mongod --shardsvr --replSet shard1rs --port 27017 --
dbpath /data/db
    ports:
      - 20003:27017
    volumes:
      - shard1s3:/data/db

volumes:
  shard1s1: {}
  shard1s2: {}
  shard1s3: {}Copy
```

The **YAML** file for the shard replica set contains specifications similar to the config server specifications. The main difference is in the **`command`** field for each replica, where the **`mongod`** command for shards is issued with the **`--shardsvr`** option. As a result, MongoDB recognizes the servers as shard instances.

4. Save and exit the file.

5. Use Docker Compose to apply the replica set configuration:

```
docker compose up -dCopy
```

The output confirms the successful creation of the Docker containers.

6. Log in to one of the replicas using the **`mongo`** command:

```
mongosh mongodb://[ip_address]:[port]Copy
```

7. Initiate the replica set with **`rs.initiate()`**:

```
rs.initiate(
  {
    _id: "shard1rs",
    members: [
      { _id : 0, host : "10.0.2.15:20001" },
      { _id : 1, host : "10.0.2.15:20002" },
      { _id : 2, host : "10.0.2.15:20003" }
    ]
  }
)Copy
```

If the initiation is successful, the output value of **`"ok"`** is **`1`**.

```
test> rs.initiate( { _id: "shard1rs", members: [ { _id: 0, host: "192.168.0.25
:20001" }, { _id: 1, host: "192.168.0.25:20002" }, { _id: 2, host: "192.168.0.
25:20003" }] } )
{ ok: 1 }
test>
```

Note: Replica set names must be unique for each shard replica set you add to the cluster.

Step 3: Start mongos Instance

A mongos instance acts as a query router, i.e., an interface between the cluster and client apps. Follow the steps below to set it up in your cluster.

1. Create a directory for your mongos configuration and navigate to it:

```
mkdir mongos && cd mongosCopy
```

2. Create a Docker Compose file:

```
nano compose.ymlCopy
```

3. Configure the mongos instance. For example, the file below creates a mongos instance and exposes it to port **30000**. The command section should contain the **--configdb** option, followed by references to the addresses of config server replicas, as in the example below:

```
services:

  mongos:
    container_name: mongos
    image: mongo
    command: mongos --configdb
cfgrs/10.0.2.15:10001,10.0.2.15:10002,10.0.2.15:10003 --bind_ip
0.0.0.0 --port 27017
    ports:
      - 30000:27017Copy
```

4. Save the file and exit.

5. Apply the configuration with **docker compose**:

```
docker compose up -dCopy
```

The output shows Docker has created the mongos instance container.

6. Check the running containers in Docker:

```
docker psCopy
```

After deploying three config server replicas, three shard replicas, and one mongos instance, the output shows seven containers based on the *mongo* image.

```
marko@phoenixnap:~/mongos$ docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED
STATUS             PORTS                                           NAMES
35758c936025   mongo:4.2.22   "docker-entrypoint.s…"   17 seconds ago
Up 15 seconds      0.0.0.0:30000->27017/tcp, :::30000->27017/tcp   mongos
eba6bdc881a1   mongo:4.2.22   "docker-entrypoint.s…"   12 minutes ago
Up 12 minutes      0.0.0.0:20002->27017/tcp, :::20002->27017/tcp   shard1s2
044aad07f6e3   mongo:4.2.22   "docker-entrypoint.s…"   12 minutes ago
Up 12 minutes      0.0.0.0:20001->27017/tcp, :::20001->27017/tcp   shard1s1
dc251a5f93f8   mongo:4.2.22   "docker-entrypoint.s…"   12 minutes ago
Up 12 minutes      0.0.0.0:20003->27017/tcp, :::20003->27017/tcp   shard1s3
8a23f9a7fa3e   mongo:4.2.22   "docker-entrypoint.s…"   About an hour ago
Up About an hour   0.0.0.0:10003->27017/tcp, :::10003->27017/tcp   configs3
19e47bca2694   mongo:4.2.22   "docker-entrypoint.s…"   About an hour ago
Up About an hour   0.0.0.0:10001->27017/tcp, :::10001->27017/tcp   configs1
6ddb53ef969f   mongo:4.2.22   "docker-entrypoint.s…"   About an hour ago
Up About an hour   0.0.0.0:10002->27017/tcp, :::10002->27017/tcp   configs2
marko@phoenixnap:~/mongos$
```

Step 4: Connect to Sharded Cluster

With all the instances up and running, the rest of the cluster configuration is performed inside the cluster. Connect to the cluster using the **mongosh** command:

```
mongosh mongodb://[mongos_ip_address]:[mongos_port]Copy
```

The MongoDB shell command prompt appears.

```
marko@phoenixnap:~$ mongosh mongodb://192.168.0.25:30000
Current Mongosh Log ID: 6756bd652e65e2c7c9e94969
Connecting to:          mongodb://192.168.0.25:30000/?directConnection=true&ap
pName=mongosh+2.3.4
Using MongoDB:          4.2.22
Using Mongosh:          2.3.4

For mongosh info see: https://www.mongodb.com/docs/mongodb-shell/

[direct: mongos] test>
```

Step 5: Add Shards to Cluster

Use the **`sh.addshard()`** method and connect the shard replicas to the cluster:

```
sh.addShard("[shard_replica_set_name]/[shard_replica_1_ip]:[port],[shard_replica_2_ip]:[port],[shard_replica_3_ip]:[port]")Copy
```

The output shows that the system successfully added the shards to the cluster. The **`"ok"`** value is **`1`**:

```
[direct: mongos] test> sh.addShard("shard1rs/192.168.0.25:20001,192.168.0.25:20002,192.168.0.25:20003")
{
  shardAdded: 'shard1rs',
  ok: 1,
  operationTime: Timestamp({ t: 1733738419, i: 6 }),
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1733738419, i: 6 }),
    signature: {
      hash: Binary.createFromBase64('AAAAAAAAAAAAAAAAAAAAAAAAAAA=', 0),
      keyId: Long('0')
    }
  }
}
[direct: mongos] test>
```

Check the status with the **`sh.status()`** method:

```
sh.status()Copy
```

The output lists the active shards in the **`shards`** section:

```
    shards:
          { "_id" : "shard1rs",  "host" : "shard1rs/192.168.0.25:20001,192.168.0.25:20002,192.168.0.25:20003",  "state" : 1,  "topologyTime" : Timestamp(1654521278, 1) }
```

Step 6: Enable Sharding for Database

Enable sharding for each included database. Use the **`sh.enableSharding()`** method followed by the database name.

```
sh.enableSharding("[database_name]")Copy
```

The example below enables sharding for the database named **`testdb`**:

```
[direct: mongos] test> sh.enableSharding("testdb")
{
        "ok" : 1,
        "$clusterTime" : {
                "clusterTime" : Timestamp(1654521652, 1),
                "signature" : {
                        "hash" : BinData(0,"AAAAAAAAAAAAAAAAAAAAAAAAAAA="),
                        "keyId" : NumberLong(0)
                }
        },
        "operationTime" : Timestamp(1654521652, 1)
}
[direct: mongos] test>
```

Note: Database workloads require high memory density and large storage capacity to perform well. Our **BMC database servers** are workload-optimized and support all major databases.

Step 7: Shard Collection

There are two ways to shard a collection in MongoDB. Both ways use the **`sh.shardCollection()`** method:

- **Range-based sharding** produces a shard key using multiple fields and creates contiguous data ranges based on the shard key values.
- **Hashed sharding** forms a shard key using a single field's hashed index.

To shard a collection using range-based sharding, specify the field to use as a shard key and set its value to **`1`**:

`sh.shardCollection("[database].[collection]", { [field]: 1 } )`Copy

```
[direct: mongos] test> sh.shardCollection("testdb.testcol", { name: 1 } )
{
        "collectionsharded" : "testdb.testcol",
        "ok" : 1,
        "$clusterTime" : {
                "clusterTime" : Timestamp(1654522900, 8),
                "signature" : {
                        "hash" : BinData(0,"AAAAAAAAAAAAAAAAAAAAAAAAAAA="),
                        "keyId" : NumberLong(0)
                }
        },
        "operationTime" : Timestamp(1654522900, 4)
}
[direct: mongos] test>
```

Use the same syntax to set up hashed sharding. This time, set the value of the field to **"hashed"**:

```
sh.shardCollection("[database].[collection]", { [field]: "hashed" } )Copy
```

```
[direct: mongos] test> sh.shardCollection("testdb.testcol1", { name: "hashed" } )
{
        "collectionsharded" : "testdb.testcol1",
        "ok" : 1,
        "$clusterTime" : {
                "clusterTime" : Timestamp(1654523068, 27),
                "signature" : {
                        "hash" : BinData(0,"AAAAAAAAAAAAAAAAAAAAAAAAAAA=),
                        "keyId" : NumberLong(0)
                }
        },
        "operationTime" : Timestamp(1654523068, 23)
}
[direct: mongos] test>
```

Warning: Once you shard a collection, you cannot unshard it.

MongoDB Sharding: Best Practices

Follow the best practices listed below to ensure an effective implementation and management of MongoDB clusters:

- **Maximize shard key selectiveness**. The shard key should effectively and evenly distribute data across shards.
- **Pay attention to data access patterns**. Minimize data movement and improve query performance by choosing a shard key that aligns with the app's data querying patterns.
- **Monitor data distribution**. Identify and address potential data skews early by regularly monitoring data distribution across shards.
- **Use range-based queries**. Leverage range-based queries on the shard key to efficiently retrieve data from specific shards.
- **Regularly review and update security configurations**. Implement appropriate security measures, such as **authentication** and authorization. Keep your sharded cluster secure by regularly reviewing and updating security configurations.
- **Simulate production workloads**. Identify and address potential performance bottlenecks by performing production workload simulations.

Conclusion

After reading this tutorial, you can deploy a sharded MongoDB cluster using Docker and Docker Compose. The article also introduced sharding as a database management technique and provided some best practice tips.

If you are interested in how MongoDB compares against popular database management solutions, read **MongoDB vs. MySQL** and **Cassandra vs. MongoDB**.