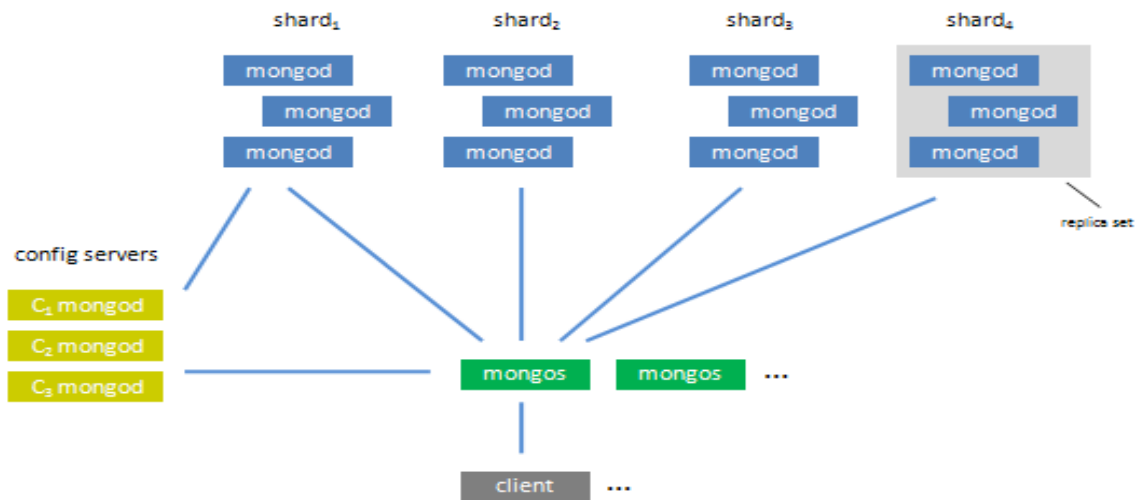
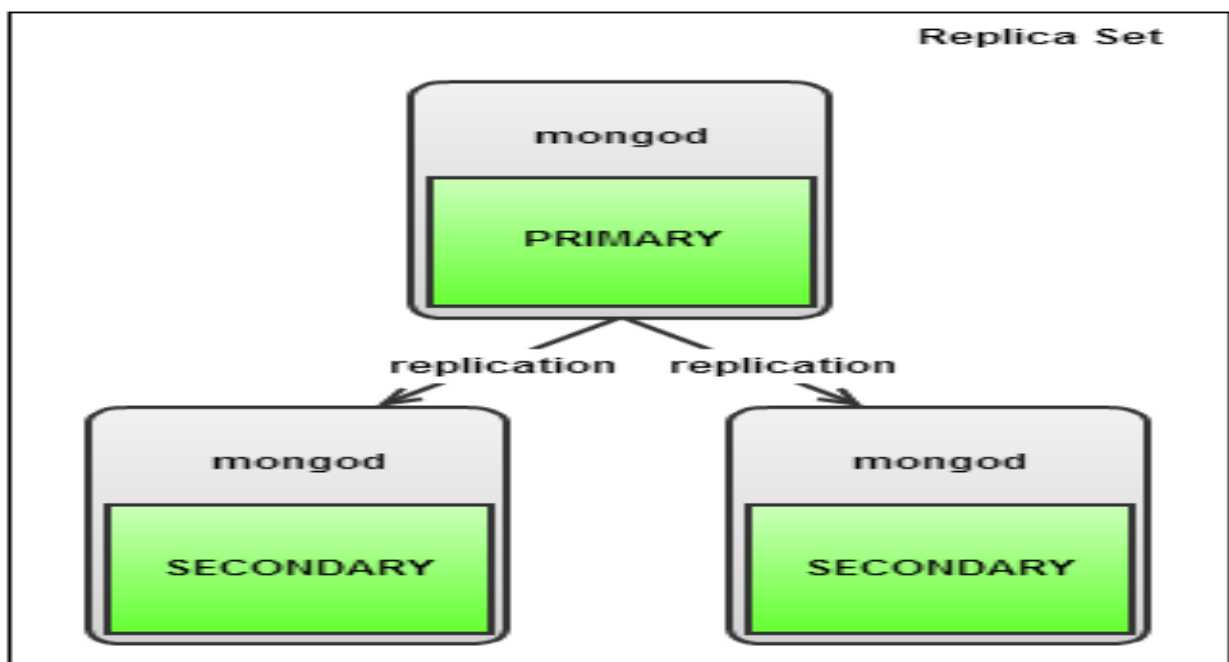

MongoDB Sharded Cluster with Replica Set



Replica Sets or Sharded Clusters?

Replica Sets are a great way to replicate MongoDB data across multiple servers and have the database automatically failover in case of server failure. Read workloads can be scaled by having clients directly connect to secondary instances. Note that master/slave MongoDB replication is not the same thing as a Replica Set, and does not have automatic failover.

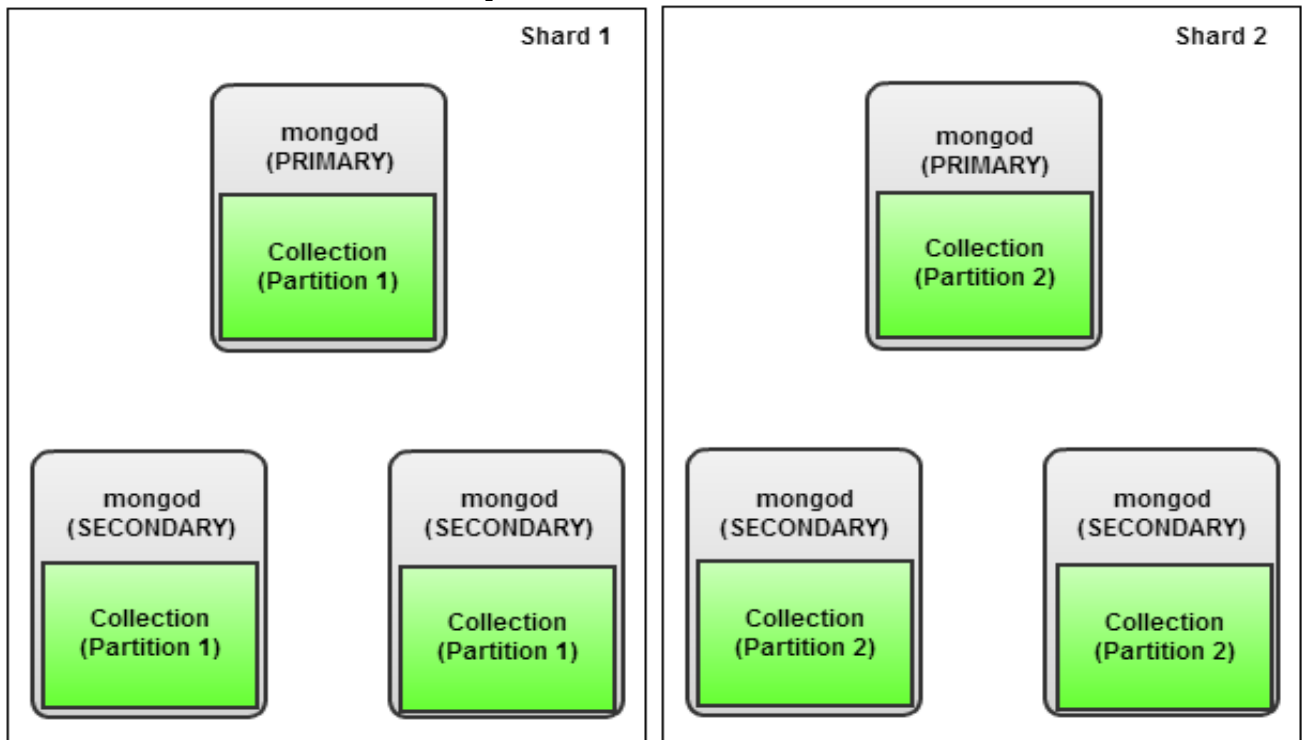
Since replication is asynchronous, the data on secondary instances might not be the latest.



Sharding is a way to split data across multiple servers. In a MongoDB Sharded Cluster, the database will handle distribution of the data and dynamically load-balance queries. So, if you have a

high write workload, then sharding is the way to go. MongoDB uses a shard key to split collections and migrate the ‘chunks’ to the correct shard.

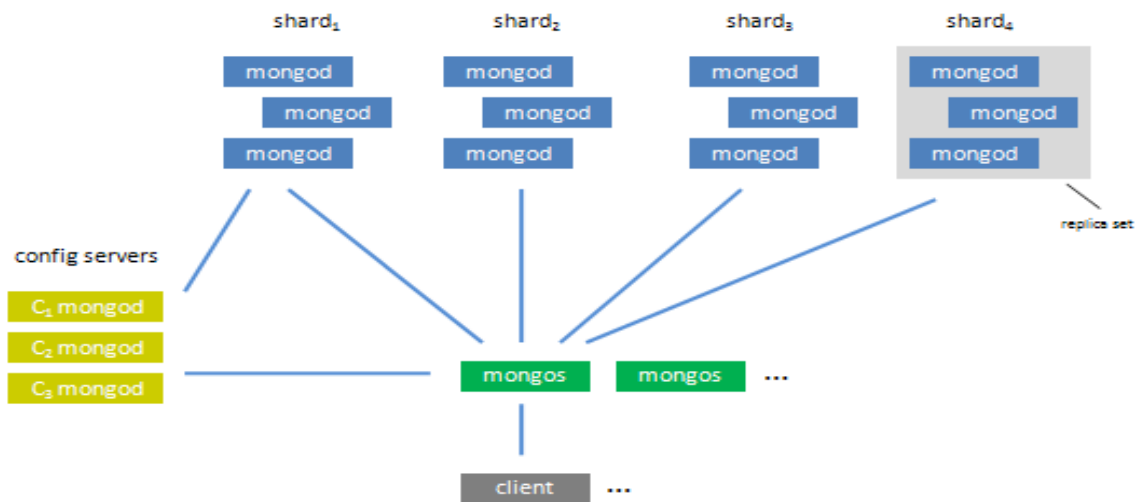
It is important to pick a good shard key and avoid “hot spots”, so that data is evenly spread between shards and writes are also evenly distributed. In the picture below you see an example with two shards, each shard consists of a replica set.



Migration from a Replica Set to a Sharded setup is pretty easy, but should not be done when the system is busy. The splitting/migration of chunks creates extra load, and can bring a busy system to a standstill.

Additional configuration and routing processes are added to manage the data and query distribution. Config servers store the meta data for the sharded cluster, and are kept consistent by using a two phase commit. Routers are the processes that clients connect to, and queries are then routed to the appropriate shard.

Sharding does add some more complexity, since there are more processes to manage. However, if your performance is degrading and tuning of your application or your existing instances are not helping, then you probably need to look into sharding.



What's the difference between sharding DB tables and partitioning them?

Partitioning is a general term used to describe **the act of breaking up your logical data elements into multiple entities** for the purpose of performance, availability, or maintainability.

Sharding is the equivalent of "horizontal partitioning". When you shard a database, you create replica's of the schema, and then divide what data is stored in each shard based on a shard key. For example, I might shard my customer database using CustomerId as a shard key - I'd store ranges 0-10000 in one shard and 10001-20000 in a different shard. When choosing a shard key, the DBA will typically look at data-access patterns and space issues to ensure that they are distributing load and space across shards evenly.

"Vertical partitioning" is the act of splitting up the data stored in one entity into multiple entities - again for space and performance reasons. For example, a customer might only have one billing address, yet I might choose to put the billing address information into a separate table with a CustomerId reference so that I have the flexibility to move that information into a separate database, or different security context, etc

To summarize - partitioning is a generic term that just means dividing your logical entities into different physical entities for performance, availability, or some other purpose. "Horizontal partitioning", or sharding, is replicating the schema, and then dividing the data based on a shard key. "Vertical partitioning" involves dividing up the schema (and the data goes along for the ride).

Links ::

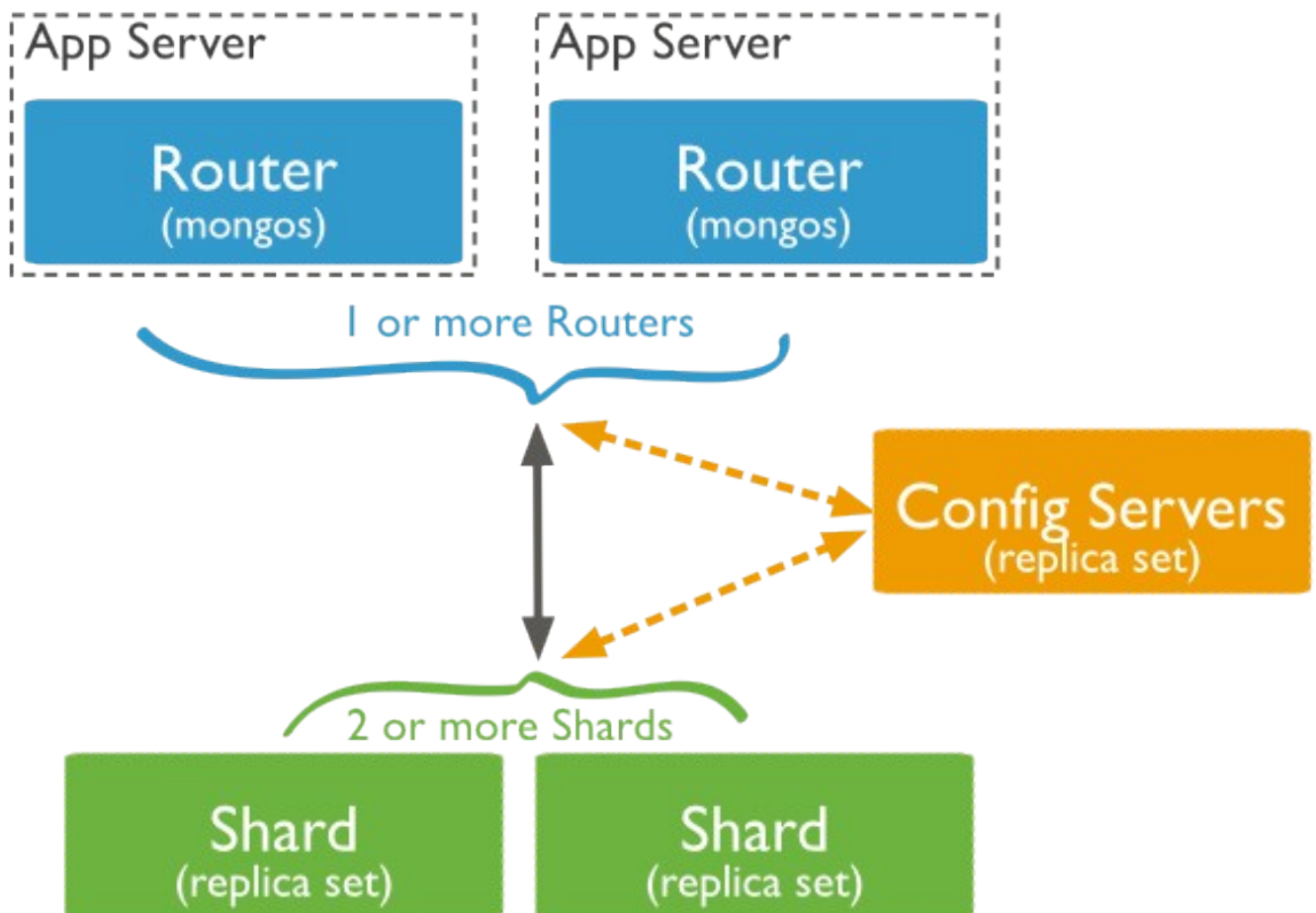
<https://www.quora.com/Whats-the-difference-between-sharding-DB-tables-and-partitioning-them>

Sharded Cluster

A MongoDB [sharded cluster](#) consists of the following components:

- [shard](#): Each shard contains a subset of the sharded data. Each shard can be deployed as a [replica set](#).
- [mongos](#): The mongos acts as a query router, providing an interface between client applications and the sharded cluster.
- [config servers](#): Config servers store metadata and configuration settings for the cluster. As of MongoDB 3.2, config servers can be deployed as a replica set.

The following graphic describes the interaction of components within a sharded cluster:



MongoDB shards data at the [collection](#) level, distributing the collection data across the shards in the cluster.

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. See [Shard Key Indexes](#).

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.

Links ::
<https://docs.mongodb.com/manual/sharding/>

MongoDB Sharding Topology

Sharding is implemented through three separate components. Each part performs a specific function:

- **Config Server:** Each production sharding implementation must contain exactly three configuration servers. This is to ensure redundancy and high availability. Config servers are used to store the metadata that links requested data with the shard that contains it. It organizes the data so that information can be retrieved reliably and consistently.
- **Query Routers:** The query routers are the machines that your application actually connects to. These machines are responsible for communicating to the config servers to figure out where the requested data is stored. It then accesses and returns the data from the appropriate shard(s).
Each query router runs the "mongos" command.
- **Shard Servers:** Shards are responsible for the actual data storage operations. In production environments, a single shard is usually composed of a replica set instead of a single machine. This is to ensure that data will still be accessible in the event that a primary shard server goes offline.

Initial Set Up (for production)

If you were paying attention above, you probably noticed that this configuration requires quite a few machines. In this tutorial, we will configure an example sharding cluster that contains:

- ***3 Config Servers (Required in production environments)***
- ***2 Query Routers (Minimum of 1 necessary)***
- ***4 Shard Servers (Minimum of 2 necessary)***

This means that you will need nine VPS instances to follow along exactly. In reality, some of these functions can overlap (for instance, you can run a query router on the same VPS you use as a config server) and you only need one query router and a minimum of 2 shard servers.

Initial Set Up (for testing)

- ***1 Config Servers (Required in production environments)***
- ***1 Query Routers (Minimum of 1 necessary)***
- ***2 Shard Servers (Minimum of 2 necessary)***

in my case ::

192.168.1.253	ubuntu-server.com	rep-master	}	shard-1
192.168.1.254	ubuntu-client.com	rep-slave	}	
192.168.1.234	biz.com	arbitrary	}	
192.168.1.18	ubuntubiz			shard-2
192.168.1.63	ubuntumukki			query
192.168.1.23	ubuntu-master.com			config

here i use one replication set as shard and another shard without replication set.

Imp :: use replication set as shared in both for production, i use only one shard as replication set due to lack of enough system

imp :: run as root

steps ::

- 1) Installation***
- 2) Hostname and FQDN***
- 3) Replication set***
- 4) Config server***
- 5) Query server***
- 6) Shard servers***
- 7) Shard-key and shard-tag***

Installation

install mongodb in all systems

```
# apt-key adv --keyserver keyserver.ubuntu.com --recv 7F0CEB10
# echo "deb http://downloads-distro.mongodb.org/repo/ubuntu-upstart dist 10gen" | tee -a
/etc/apt/sources.list.d/10gen.list
# apt-get -y update
# apt-get -y install mongodb-10gen
```

links ::

<https://www.digitalocean.com/community/tutorials/how-to-install-mongodb-on-ubuntu-12-04>

Hostname and FQDN

create hostname and FQDN and add this name in /etc/hosts of all systems
create hostname and FQDN same to avoid conflict.

nano /etc/hosts

127.0.1.1 ubuntu-master.com

nano /etc/hostname

ubuntu-master.com

do this for all systems with appropriate name

after that add all hostname to /etc/hosts of all systems

nano /etc/hosts

```
ubuntu@ubuntu-master: ~  
GNU nano 2.2.6 File: /etc/hosts  
  
127.0.0.1      localhost  
127.0.1.1      ubuntu-master.com  
# The following lines are desirable for IPv6 capable hosts  
::1      ip6-localhost ip6-loopback  
fe00::0 ip6-localnet  
ff00::0 ip6-mcastprefix  
ff02::1 ip6-allnodes  
ff02::2 ip6-allrouters  
  
#192.168.1.253 ubuntu-server.com server  
#192.168.1.254 ubuntu-client.com client  
[  
  
192.168.1.253 ubuntu-server.com  
192.168.1.254 ubuntu-client.com  
192.168.1.63      ubuntuukki  
192.168.1.234      biz.com  
192.168.1.18      ubuntuibz  
192.168.1.23      ubuntu-master.com  
  
^G Get Help      ^O WriteOut      ^R Read File      ^Y Prev Page      ^K Cut Text      ^C Cur Pos  
^X Exit          ^J Justify      ^W Where Is      ^V Next Page      ^U UnCut Text      ^T To Spell
```

Replication set

Replication set is configuring for shared-1 if you are interesting to use single system as shared like shard-2 skip this step

Steps :

- a) Install and configure mongodb master and slave node*
- b) Arbiter node configure for failover*
- c) testing*

Difference between “master-slave replication” and “replication set” in mongodb

The difference between a replication set and master-slave replication is that a replication set has an intrinsic automatic failover mechanism in case the primary member becomes unavailable.

Install and configure mongodb master and slave node

my configuration ::

192.168.1.253 ubuntu-server.com
192.168.1.254 ubuntu-client.com
192.168.1.234 biz.com

in initial condition;

<i>192.168.1.253</i>	<i>ubuntu-server.com</i>	<i>>></i>	<i>Primary Server</i>
<i>192.168.1.254</i>	<i>ubuntu-client.com</i>	<i>>></i>	<i>Secondary Server</i>
<i>192.168.1.234</i>	<i>biz.com</i>	<i>>></i>	<i>Arbiter</i>

<i>master node</i>	<i>>></i>	<i>primary member</i>
<i>slave node</i>	<i>>></i>	<i>secondary member</i>
<i>arbiter node</i>	<i>>></i>	<i>Arbiter</i>

Primary member: The primary member is the default access point for transactions with the replication set. It is the only member that can accept write operations.

Each replication set can have only one primary member at a time. This is because replication happens by copying the primary's "oplog" (operations log) and repeating the changes on the secondary's dataset. Multiple primaries accepting write operations would lead to data conflicts.

Secondary members: A replication set can contain multiple secondary members. Secondary members reproduce changes from the oplog on their own data.

Although by default applications will query the primary member for both read and write operations, you can configure your setup to read from one or more of the secondary members. A secondary member can become the primary if the primary goes offline or steps down.

Arbiter: An arbiter is an optional member of a replication set that does not take part in the actual replication process. It is added to the replication set to participate in only a single, limited function: to act as a tie-breaker in elections.

Consider you have 2 MongoDB nodes. One act as primary and the other once act as secondary node. If any one of the node goes down then MongoDB cannot able to choose a primary node by itself because MongoDB needs majority of members (at least two active nodes) to choose a primary node in a multi node replica set. As a result of this all write operations are failed and a lock occurs. In this scenario if an arbiter exists then the arbiter will vote for the available node to become primary.

An arbiter does **not** have a copy of data set and **cannot** become a primary. Replica sets may have arbiters to add a vote in [elections of for primary](#). Arbiters *always* have exactly 1 vote election, and thus allow replica sets to have an uneven number of members, without the overhead of a member that replicates data.

Do not run an arbiter on systems that also host the primary or the secondary members of the replica set.

In the event that the primary member becomes unavailable, an automated election process happens among the secondary nodes to choose a new primary. If the secondary member pool contains an even number of nodes, this could result in an inability to elect a new primary due to a voting impasse. The arbiter votes in these situations to ensure a decision is reached.

Set Up DNS Resolution

In order for our MongoDB instances to communicate with each other effectively, we will need to configure our machines to resolve the proper hostname for each member. You can either do this by [configuring subdomains for each replication member](#) or through editing the /etc/hosts file on each computer.

nano /etc/hosts

```
192.168.1.253  ubuntu-server.com
192.168.1.254  ubuntu-client.com
192.168.1.63   ubuntu-mukki
192.168.1.234  biz.com
192.168.1.18   ubuntu-biz
192.168.1.23   ubuntu-master.com
```

do the above steps on all machines.

Prepare for Replication in the MongoDB Configuration File

The first thing we need to do to begin the MongoDB configuration is stop the MongoDB process on each server.

On each sever, type:

```
# service mongod stop
```

may be sometime this will work but status will showing that service is stopped., but don't worry, it will work.

Now, we need to configure a directory that will be used to store our data. Create a directory with the following command:

```
# mkdir -p /home/mongo/mongo-metadata
```

Now that we have the data directory created, we can modify the configuration file to reflect our new replication set configuration:

```
# nano /etc/mongodb.conf
```

In this file, we need to specify a few parameters. First, adjust the dbpath variable to point to the directory we just created:

```
dbpath=/home/mongo/mongo-metadata
```

Remove the comment from in front of the port number specification to ensure that it is started on the default port:

```
port = 27017
```

Towards the bottom of the file, remove the comment form in front of the replSet parameter. Change the value of this variable to something that will be easy to recognize for you.

```
replSet = rs0
```

Finally, you should make the process fork so that you can use your shell after spawning the server instance. Add this to the bottom of the file:

```
fork = true
```

```
root@ubuntu-server: ~
root@ubuntu-server: ~
root@ubuntu-client: ~
root@biz: ~
GNU nano 2.2.6 File: /etc/mongodb.conf

# mongodb.conf
# Where to store the data.
# Note: if you run mongodb as a non-root user (recommended) you may
# need to create and set permissions for this directory manually,
# e.g., if the parent directory isn't mutable by the mongodb user.
dbpath=/home/mongo/mongo-metadata

#where to log
logpath=/var/log/mongodb/mongodb.log
logappend=true

port = 27017

# Disables write-ahead journaling
# nojournal = true

# Enables periodic logging of CPU utilization and I/O wait
#cpu = true

# Turn on/off security. Off is currently the default
#noauth = true
#auth = true

# Verbose logging output.
#verbose = true

# Inspect all client data for validity on receipt (useful for
# developing drivers)
#objcheck = true

# Enable db quota management

^G Get Help      ^O WriteOut      ^R Read File     ^Y Prev Page    ^K Cut Text      ^C Cur Pos
^X Exit          ^J Justify       ^W Where Is      ^V Next Page    ^U UnCut Text    ^T To Spell
```

Save and close the file. Start the replication member by issuing the following command:

```
# mongod --config /etc/mongodb.conf
```

These steps must be repeated on each member of the replication set.

Start the Replication Set and Add Members

Primary server Configuration :

Now that you have configured each member of the replication set and started the mongod process on each machine, you can initiate the replication and add each member. On one of your members, type:

```
# mongo
```

This will give you a MongoDB prompt for the current member. Start the replication set by entering:

```
> rs.initiate()
```

This will initiate the replication set and add the server you are currently connected to as the first member of the set. You can see this by typing:

```
> rs.conf()
```

Now, you can add the additional nodes to the replication set by referencing the hostname you gave them in the /etc/hosts file:

```
> rs.add("ubuntu-client.com")
```

Do this for each of your remaining replication members. Your replication set should now be up and running.

Now add the arbiter server with following command

```
# rs.addArb("biz.com")
```

This will initiate the replication set and add the server you are currently connected to as the first member of the set. You can see this by typing:

```
> rs.status()
```

```
rs0:PRIMARY> rs.status()
```

```
{
  "set" : "rs0",
  "date" : ISODate("2016-11-21T10:23:20Z"),
  "myState" : 1,
  "members" : [
    {
      "_id" : 0,
      "name" : "ubuntu-server.com:27017",
      "health" : 1,
      "state" : 1,
      "stateStr" : "PRIMARY",
      "uptime" : 7901,
      "optime" : Timestamp(1479723778, 1),
      "optimeDate" : ISODate("2016-11-21T10:22:58Z"),
      "self" : true
    },
    {
      "_id" : 1,
      "name" : "ubuntu-client.com:27017",
      "health" : 1,
      "state" : 2,
      "stateStr" : "SECONDARY",
      "uptime" : 56,
      "optime" : Timestamp(1479723778, 1),
      "optimeDate" : ISODate("2016-11-21T10:22:58Z"),
      "lastHeartbeat" : ISODate("2016-11-21T10:23:18Z"),
      "lastHeartbeatRecv" : ISODate("2016-11-21T10:23:20Z"),
      "pingMs" : 5,
      "lastHeartbeatMessage" : "syncing to: ubuntu-server.com:27017",
      "syncingTo" : "ubuntu-server.com:27017"
    },
    {

```

```

        "_id" : 2,
        "name" : "biz.com:27017",
        "health" : 1,
        "state" : 7,
        "stateStr" : "ARBITER",
        "uptime" : 1984,
        "lastHeartbeat" : ISODate("2016-11-21T10:23:19Z"),
        "lastHeartbeatRecv" : ISODate("2016-11-21T10:23:19Z"),
        "pingMs" : 8
      },
    ],
    "ok" : 1
  }

```

Now restart the MongoDB instance in all the three servers

```
# service mongod restart
```

do this on all servers

Testing Replication and Failover

Replication Test :

in Primary server :

create database,

```
> use <test1>
```

add some content to this database

```
> db.test1.insert({item: "name1"})
```

check whether the database created or not,

```
> show dbs
```

Do all this steps in Secondary server, it will not create database, (only Read permission)

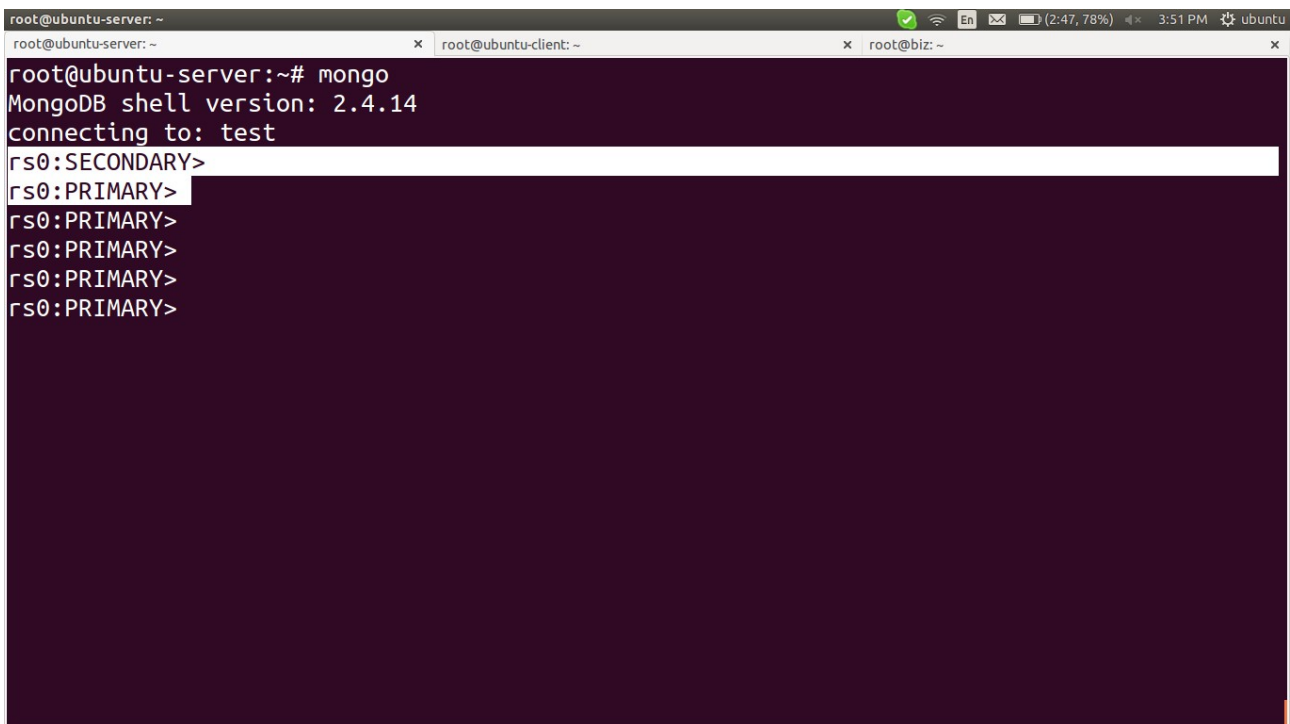
Failover test :

in primary server :

stop the service or break the network connection to that server

after sometime check in Secondary server by enter, it will shows that it changed from Secondary to primary

in secondary server :



```
root@ubuntu-server: ~  
root@ubuntu-server: ~ x root@ubuntu-client: ~ x root@biz: ~ x  
root@ubuntu-server:~# mongo  
MongoDB shell version: 2.4.14  
connecting to: test  
rs0:SECONDARY>  
rs0:PRIMARY>  
rs0:PRIMARY>  
rs0:PRIMARY>  
rs0:PRIMARY>  
rs0:PRIMARY>
```

It can also visible in Arbitary node,

> rs.conf()

Links ::

failover with arbitary ::

<https://docs.mongodb.com/v3.0/core/replica-set-arbiter/>

<http://www.congruentsolutions.com/mongodb-replication-approach-setup-using-arbiter/>

installation and configuration::

<https://www.digitalocean.com/community/tutorials/how-to-install-mongodb-on-ubuntu-12-04>

<https://www.digitalocean.com/community/tutorials/how-to-implement-replication-sets-in-mongodb-on-an-ubuntu-vps>

Config server

192.168.1.23 ubuntu-master.com

The first components that must be set up are the configuration servers. These must be online and operational before the query routers or shards can be configured. Log into your first configuration server as root.

The first thing we need to do is create a data directory, which is where the configuration server will store the metadata that associates location and content:

```
# mkdir /mongo-metadata
```

Now, we simply have to start up the configuration server with the appropriate parameters. The service that provides the configuration server is called mongod. The default port number for this component is 27019.

We can start the configuration server with the following command:

```
# mongod --configsvr --dbpath /mongo-metadata --port 27019
```

The server will start outputting information and will begin listening for connections from other components.

Repeat this process exactly on the other two configuration servers. The port number should be the same across all three servers.

Query server

192.168.1.63 *ubuntumukki*

At this point, you should have all three of your configuration servers running and listening for connections. They must be operational before continuing.
Log into your first query router as root.

The first thing we need to do is stop the mongod process on this instance if it is already running. The query routers use data locks that conflict with the main MongoDB process:

```
# service mongod stop
```

or

```
# ps-aux | grep mongo
```

kill the mongo related process
eg ::

```
# killall mongod
```

Next, we need to start the query router service with a specific configuration string. The configuration string must be exactly the same for every query router you configure (including the order of arguments). It is composed of the address of each configuration server and the port number it is operating on, separated by a comma.

The query router service is called mongos. The default port number for this process is 27017 (but the port number in the configuration refers to the configuration server port number, which is 27019 by default).

The end result is that the query router service is started with a string like this:

```
# mongos -- configdb ubuntu-master.com:27019
```

if there are more config server use,

```
# mongos -- configdb config0.example.com:27019,config1.example.com:27019
```

Your first query router should begin to connect to the three configuration servers. Repeat these steps on the other query router. Remember that the mongod service must be stopped prior to typing in the command.

Also, keep in mind that the **exact** same command must be used to start each query router. Failure to do so will result in an error.

Shard servers

```
192.168.1.253  ubuntu-server.com rep-master  }    shard-1
192.168.1.254  ubuntu-client.com rep-slave   }
192.168.1.234  biz.com           arbitary   }
```

```
192.168.1.18  ubuntubiz                               shard-2
```

To actually add the shards to the cluster, we will need to go through the query routers, which are now configured to act as our interface with the cluster. We can do this by connecting to *any* of the query routers from any system have mongodb, like this:

```
# mongo --host ubuntumukki --port 27017
```

if it is not able to login from remote system, mostly because of iptables,

```
# iptables -A INPUT -p tcp --destination-port 27017 -m state --state NEW,ESTABLISHED -j ACCEPT
```

```
# iptables -A OUTPUT -p tcp --source-port 27017 -m state --state ESTABLISHED -j ACCEPT
```

This will connect to the appropriate query router and open a mongo prompt. We will add all of our shard servers from this prompt.

If you are configuring a production cluster, complete with replication sets, you have to instead specify the replication set name and a replication set member to establish each set as a distinct shard. The syntax would look something like this:

```
# sh.addShard( "rep_set_name/hostname:27017,hostname:27017" )
```

To add our first shard, type:

```
# sh.addShard( "ubuntubiz:27017" )
# sh.addShard( "rs0/ubuntu-server.com:27017,ubuntu-client.com:27017" )
```

Enable Sharding on the Database Level

We will enable sharding first on the database level. To do this, we will create a test database called (appropriately) test_db.

To create this database, we simply need to change to it. It will be marked as our current database and created dynamically when we first enter data into it:

if you logout then log in again to query server

```
# mongo --host ubuntumukki --port 27017
```

```
> use test_db
```

We can check that we are currently using the database we just created by typing:

```
> db
```

We can see all of the available databases by typing:

```
> show dbs
```

You may notice that the database that we just created does not show up. This is because it holds no data so it is not quite real yet.

We can enable sharding on this database by issuing this command:

```
> sh.enableSharding("test_db")
```

Again, if we enter the show dbs command, we will not see our new database. However, if we switch to the config database which is generated automatically, and issue a find() command, our new database will be returned:

```
> use config
> db.databases.find()
```

```
{ "_id" : "admin", "partitioned" : false, "primary" : "config" }
{ "_id" : "test_db", "partitioned" : true, "primary" : "shard0000" }
```

Enable Sharding on the Collections Level

Now that our database is marked as being available for sharding, we can enable sharding on a specific collection.

At this point, we need to decide on a sharding strategy. Sharding works by organizing data into different categories based on a specific field designated as the shard key in the documents it is storing. It puts all of the documents that have a matching shard key on the same shard.

For instance, if your database is storing employees at a company and your shard key is based on favorite color, MongoDB will put all of the employees with blue in the favorite color field on a single shard. This can lead to disproportional storage if everybody likes a few colors.

A better choice for a shard key would be something that's guaranteed to be more evenly distributed. For instance, in a large company, a birthday (month and day) field would probably be fairly evenly distributed.

In cases where you're unsure about how things will be distributed, or there is no appropriate field, you can create a "hashed" shard key based on an existing field. This is what we will be doing for our data.

We can create a collection called test_collection and hash its "id" field. Make sure we're using our testdb database and then issue the command:

```
> use test_db
> db.test_collection.ensureIndex( { _id : "hashed" } )
```

We can then shard the collection by issuing this command:

```
> sh.shardCollection("test_db.test_collection", { "_id": "hashed" })
```

Insert Test Data into the Collection

We can see our sharding in action by using a loop to create some objects. This [loop comes directly from the MongoDB website](#) for generating test data.

We can insert data into the collection using a simple loop like this:

```
> use test_db
> for (var i = 1; i <= 500; i++) db.test_collection.insert( { x : i } )
```

This will create 500 simple documents (only an ID field and an "x" field containing a number) and distribute them among the different shards. You can see the results by typing:

```
> use test_db
> db.test_collection.find()
```

```
{ "_id" : ObjectId("529d082c488a806798cc30d3"), "x" : 6 }
{ "_id" : ObjectId("529d082c488a806798cc30d0"), "x" : 3 }
{ "_id" : ObjectId("529d082c488a806798cc30d2"), "x" : 5 }
{ "_id" : ObjectId("529d082c488a806798cc30ce"), "x" : 1 }
{ "_id" : ObjectId("529d082c488a806798cc30d6"), "x" : 9 }
{ "_id" : ObjectId("529d082c488a806798cc30d1"), "x" : 4 }
{ "_id" : ObjectId("529d082c488a806798cc30d8"), "x" : 11 }
...
```

To get more values, type:

```
> it
```

```
{ "_id" : ObjectId("529d082c488a806798cc30cf"), "x" : 2 }
{ "_id" : ObjectId("529d082c488a806798cc30dd"), "x" : 16 }
{ "_id" : ObjectId("529d082c488a806798cc30d4"), "x" : 7 }
{ "_id" : ObjectId("529d082c488a806798cc30da"), "x" : 13 }
{ "_id" : ObjectId("529d082c488a806798cc30d5"), "x" : 8 }
{ "_id" : ObjectId("529d082c488a806798cc30de"), "x" : 17 }
{ "_id" : ObjectId("529d082c488a806798cc30db"), "x" : 14 }
{ "_id" : ObjectId("529d082c488a806798cc30e1"), "x" : 20 }
...
```

To get information about the specific shards, you can type:

```
> sh.status()
```

```
*****
```

links ::

<https://www.digitalocean.com/community/tutorials/how-to-create-a-sharded-cluster-in-mongodb-using-an-ubuntu-12-04-vps>

<http://severalnines.com/blog/turning-mongodb-replica-set-sharded-cluster>

```
*****
```

Shard-key and shard-tag

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. See [Shard Key Indexes](#).

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.

Example ::

create a shared-key for routing datas of collections to particular shared with respect to the country field in that collections

```
# mongo --host ubuntuukki --port 27017
```

To get information about the specific shards, you can type:

```
> sh.status()
```

consider the name of shards are

```
shard0000  
shard0001
```

if you are doing replication set then the name may like rs0, rs1 (your replication_set_name)
adding tag to the shards

```
tags >> USA, EU
```

```
> sh.addShardTag("shard0000", "USA")  
> sh.addShardTag("shard0001", "EU")
```

```
> use test_collection
```

```
> sh.enableSharding("test_collection")
```

check the database if sharding or not ("partitioned" : true)

```
> use config
> db.databases.find()
```

```
{ "_id" : "admin", "partitioned" : false, "primary" : "config" }
{ "_id" : "test_collection", "partitioned" : true, "primary" : "shard0000" }
```

or

```
> db.printShardingStatus()
```

```
--- Sharding Status ---
```

```
sharding version: {
  "_id" : 1,
  "version" : 3,
  "minCompatibleVersion" : 3,
  "currentVersion" : 4,
  "clusterId" : ObjectId("555df7f4f6506e6ba07e1f20")
}
shards:
{ "_id" : "shard0000", "host" : "127.0.0.1:27017", "tags" : [ "USA" ] }
{ "_id" : "shard0001", "host" : "127.0.0.1:27018", "tags" : [ "EU" ] }
databases:
{ "_id" : "admin", "partitioned" : false, "primary" : "config" }
{ "_id" : "test_collection", "partitioned" : true, "primary" : "shard0000" }
```

```
> use admin
```

Then add shard key to collection we'll use

```
> db.runCommand({shardCollection: "test_collection.items", key: {"country": 1}})
```

Add tag ranges for our shards:

```
> sh.addTagRange("test_collection.items", {country: "MX"}, {country: "USA"}, "USA")
> sh.addTagRange("test_collection.items", {country: "ES"}, {country: "GER"}, "EU")
```

check whether the newly created database is shard to both shard servers

```
> db.printShardingStatus()
```

or

```
> use config
> db.databases.find()
```

Ok, let's add elements to collection

> use test_collection

```
for(var i=0; i<1000; i++)
{
  db.items.insert({"model": "One", "country": "EU"});
  db.items.insert({"model": "iPhone", "country": "US"});
}
```

> use test_collection

```
for(var i=0; i<10; ++i)
{
  db.items.insert({"model": "X", "country": "EU"});
}
```

See the result:

> use test_collection

> db.items.stats()

```
{
  "sharded" : true,
  "userFlags" : 1,
  "ns" : "test_collection.items",
  "count" : 2010,
  "numExtents" : 6,
  "size" : 225120,
  "storageSize" : 344064,
  "totalIndexSize" : 155344,
  "indexSizes" : {
    "_id_" : 81760,
    "country_1" : 73584
  },
  "avgObjSize" : 112,
  "nindexes" : 2,
  "nchunks" : 3,
  "shards" : {
    "shard0000" : {
      "ns" : "test_collection.items",
      "count" : 1000,
      "size" : 112000,
      "avgObjSize" : 112,
      "numExtents" : 3,
      "storageSize" : 172032,
      "nindexes" : 2,
      "lastExtentSize" : 131072,
      "paddingFactor" : 1,
      "userFlags" : 1,

```

```

        "totalIndexSize" : 81760,
        "indexSizes" : {
            "_id_" : 40880,
            "country_1" : 40880
        },
        "ok" : 1
    },
    "shard0001" : {
        "ns" : "test_collection.items",
        "count" : 1010,
        "size" : 113120,
        "avgObjSize" : 112,
        "numExtents" : 3,
        "storageSize" : 172032,
        "nindexes" : 2,
        "lastExtentSize" : 131072,
        "paddingFactor" : 1,
        "userFlags" : 1,
        "totalIndexSize" : 73584,
        "indexSizes" : {
            "_id_" : 40880,
            "country_1" : 32704
        },
        "ok" : 1
    }
},
"ok" : 1
}

```

Links ::

concepts and commands

<https://docs.mongodb.com/manual/sharding/>

<https://docs.mongodb.com/manual/core/sharding-shard-key/>

configuration

<http://stackoverflow.com/questions/30415387/mongodb-stores-data-only-to-primary-shard-in-case-choosing-tagging-shard-key>

links ::

mongodb installation ::

<https://www.digitalocean.com/community/tutorials/how-to-install-mongodb-on-ubuntu-12-04>

replication set configuration ::

<https://www.digitalocean.com/community/tutorials/how-to-implement-replication-sets-in-mongodb-on-an-ubuntu-vps>

arbiter configuration ::

<http://www.congruentsolutions.com/mongodb-replication-approach-setup-using-arbiter/>

partition and sharding ::

<https://www.quora.com/Whats-the-difference-between-sharding-DB-tables-and-partitioning-them>

sharding configuration ::

<https://www.digitalocean.com/community/tutorials/how-to-create-a-sharded-cluster-in-mongodb-using-an-ubuntu-12-04-vps>

<http://severalnines.com/blog/turning-mongodb-replica-set-sharded-cluster>

iptables ::

<http://askubuntu.com/questions/724642/unable-to-connect-to-mongo-on-remote-server>

shard-key concept ::

<https://docs.mongodb.com/manual/core/sharding-shard-key/>

<https://docs.mongodb.com/manual/sharding/>

shard-key configuration ::

<http://stackoverflow.com/questions/30415387/mongodb-stores-data-only-to-primary-shard-in-case-choosing-tagging-shard-key>

shard-tag ::

<http://dbversity.com/tag-aware/>
