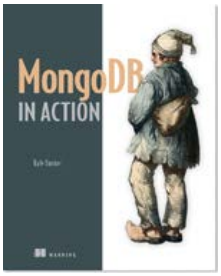
MANNING PUBLICATIONS



MongoDB in Action
By Kyle Banker

MongoDB has been designed to support sharding from the start. In this article, based on chapter 9 of MongoDB in Action, author Kyle Banker demonstrates building a sample cluster to host data from a massive Google Docs-like application.

You may also be interested in…

A Sample Shard Cluster

The best way to get a handle on sharding is to see how it works in action. Fortunately, it's possible to set up a sharded cluster on a single machine, and that's exactly what we're going to do now.[1] We'll then simulate the behavior of a sample cloud-based spreadsheet application. Along the way, we'll examine the global shard configuration and see first-hand how data is partitioned based on the shard key.

Setup

There are two steps involved in setting up a shard cluster. The first is to start all the *mongod* and *mongos* processes that comprise it. The second step, certainly the easier of the two, is to run the various commands that initiate the cluster. The shard cluster we're going to build here will consist of two shards and three config servers. We'll also start a single *mongos* to communicate with the cluster. Figure 1 shows a map of all the processes that we'll be launching with their port numbers in parentheses.

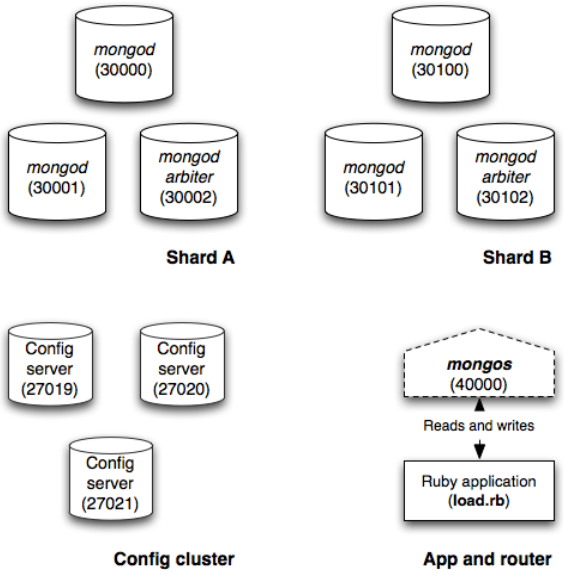


Figure 1 A map of processes comprising our sample shard cluster

[1] The idea is that we can run every mongod and mongos process on a single machine for testing.

We'll be running quite a few commands to get the cluster up and running, so if you ever find yourself losing the forest for the trees, refer back to this figure.

Starting the sharding components

Let's start by creating the data directories for the two replica sets that will serve as our shards.

```
mkdir /data/rs-a-1
mkdir /data/rs-a-2
mkdir /data/rs-a-3
mkdir /data/rs-b-1
mkdir /data/rs-b-2
mkdir /data/rs-b-3
```

Now we'll start each *mongod*. Because we're starting so many processes, we're going to specify a *logpath* for each process so that each one can be run in the background. Thus, we'll use the `--fork` options to keep from having to open too many terminal windows.[2] Here are the commands for starting the first replica set.

```
mongod --shardsvr --replSet shard-a --dbpath /data/rs-a-1 --port 30000 --logpath /data/rs-a-
mongod --shardsvr --replSet shard-a --dbpath /data/rs-a-2 --port 30001 --logpath /data/rs-a-
mongod --shardsvr --replSet shard-a --dbpath /data/rs-a-3 --port 30002 --logpath /data/rs-a-
```

And these will start the second one:

```
mongod --shardsvr --replSet shard-b --dbpath /data/rs-b-1 --port 30100 --logpath /data/rs-b-
mongod --shardsvr --replSet shard-b --dbpath /data/rs-b-2 --port 30101 --logpath /data/rs-b-
mongod --shardsvr --replSet shard-b --dbpath /data/rs-b-3 --port 30102 --logpath /data/rs-b-
```

As usual, we need to initiate each replica set. Connect to each one individually, create the config document, and then run `rs.initiate(config)`. The first should look like this:

```
$ mongo localhost:30000
> config = {_id: "shard-a",
 members: [{_id: 0, host:"localhost:30000"},
           {_id: 1, host:"localhost:30001"},
           {_id: 2, host:"localhost:30002", arbiterOnly: true}]}
> rs.initiate(config)
```

Initiating the second replica set is similar:

```
$ mongo localhost:30100
> config = {_id: "shard-b",
members: [{_id: 0, host:"localhost:30100"},
          {_id: 1, host:"localhost:30101"},
          {_id: 2, host:"localhost:30102", arbiterOnly: true}]}
rs.initiate(config)
```

Finally, verify that both replica sets are online by running the `rs.status()` command from the shell on each one. If everything checks out, you're ready to start the config servers.[3] Now we create each config server's data directory and then start a mongod for each one using the `configsvr` option.

```
$ mkdir /data/config-1 $ mongod --configsvr --dbpath /data/config-1 --logpath /data/config-
1.log --fork

$ mkdir /data/config-2 $ mongod --configsvr --dbpath /data/config-2 --logpath /data/config-
2.log --fork

$ mkdir /data/config-3 $ mongod --configsvr --dbpath /data/config-3 --logpath /data/config-
3.log --fork
```

Ensure that each config server is up and running by connecting with the shell or by printing the log file and verifying that each process is listening on the configured port. If you check the log of any one config server, you should see something like this:

```
Wed Mar 2 15:43:28 [initandlisten] waiting for connections on port 27020
Wed Mar 2 15:43:28 [websvr] web admin interface listening on port 28020
```

If each config server is running, you can go ahead and start the *mongos*. The *mongos* must be started with the *configdb* option, which takes a comma-separated list of config database addresses.[4]

```
$ mongos --port 40000 --configdb localhost:27019,localhost:27020,localhost:27021
  --logpath /data/mongos.log --fork
```

---

[2] If you're running Windows, note that fork won't work for you. Since you'll have to open a new terminal window for each process, it's best that you omit the logpath option as well.

[3] Again, if running on Windows, omit the --fork and --logpath options, and start each mongod in a new window.

[4] Be careful not to put spaces between the config server addresses when specifying them.

Configuring the cluster

You have all the sharding components in place. Now it's time to configure the cluster. Start by connecting to the *mongos*. We're going to be entering a series of configuration commands, beginning with the `addshard` command. This command takes the name of a replica set followed by the addresses of two or more seed nodes for connecting. Here, we specify the two replica sets we created along with the addresses of the two non-arbiter members of each set.

```
$ mongo localhost:40000
> use admin
switched to db admin
> db.runCommand({addshard: "shard-a/localhost:30000,localhost:30001"})
  { "shardAdded" : "shard-a", "ok" : 1 }
> db.runCommand({addshard: "shard-b/localhost:30100,localhost:30101"})
  { "shardAdded" : "shard-b", "ok" : 1 }
```

If successful, the command response will include the name of the shard added. We can examine the config database's *shards* collection to see the effect of our work.

```
> db.getSiblingDB("config").shards.find()
{ "_id" : "shard-a", "host" : "shard-a/localhost:30000,localhost:30001" }
{ "_id" : "shard-b", "host" : "shard-b/localhost:30100,localhost:30101" }
```

The `listshards` command will return the same information:

```
> use admin
> db.runCommand({listshards: 1})
```

While we're on the topic of reporting on sharding configuration, the shell's `db.printShardingStatus()` method will nicely summarize the cluster. Go ahead and try running it now.

The next configuration step is to enable sharding on a database. This is a prerequisite for sharding a particular collection. Our application's database will be called "cloud-docs," and so we enable sharding like so:

```
> db.runCommand({enablesharding: "cloud-docs"})
{ "ok" : 1 }
```

Like before, we can check the config data to see the change we just made. The config database holds a collection called *databases* that contains a list of databases and specifies their primary shard location and whether they're partitioned (in other words, whether sharding is enabled).

```
> db.getSiblingDB("config").databases.find()
{ "_id" : "admin", "partitioned" : false, "primary" : "config" }
{ "_id" : "cloud-docs", "partitioned" : true, "primary" : "shard-a" }
```

Now all we need to do is shard the *spreadsheets* collection. We're going to use the compound shard key `{username: 1, _id: 1}` because it's good for distributing data and makes it easy to view and comprehend chunk ranges.

```
> db.runCommand({shardcollection: "cloud-docs.spreadsheets", key: {username: 1, _id: 1}})
{ "collectionsharded" : "cloud-docs.spreadsheets", "ok" : 1 }
```

Again, we can verify the configuration by checking the config database for sharded collections.

```
> db.getSiblingDB("config").collections.findOne()
{
  "_id" : "cloud-docs.spreadsheets",
  "lastmod" : ISODate("1970-01-16T00:50:07.268Z"),
  "dropped" : false,
  "key" : {
    "username" : 1,
    "_id" : 1
  },
  "unique" : false
}
```

This sharded collection definition looks like an index definition, particularly with the *unique* key. In fact, when you shard an empty collection, an index corresponding to the shard key will be created on each shard.[5] Verify this for yourself by connecting directly to a shard and running the `getIndexes()` method. Here we connect to our first shard, and the output is as expected: the shard key index is there.

```
$ mongo localhost:30000
> use cloud-docs
```

[5] If you're sharding an existing collection, you'll have to create an index corresponding to the shard key before you run the "shardcollection" command.

```
> db.spreadsheets.getIndexes()
[
  {
    "name" : "_id_",
    "ns" : "cloud-docs.spreadsheets",
    "key" : {
      "_id" : 1
    },
    "v" : 0
  },
  {
    "ns" : "cloud-docs.spreadsheets",
    "key" : {
      "username" : 1,
      "_id" : 1
    },
    "name" : "username_1__id_1",
    "v" : 0
  }
]
```

Once you've sharded the collection, then, believe it or not, sharding is finally ready to go. You can now write the cluster, and data will distribute. We'll see how that works in the next section.

Writing to a sharded cluster

We're going to write to our sharded collection so that we can observe the formation and movement of chunks, which is the essence of MongoDB's sharding. Our sample documents, each representing a single spreadsheet, look like this.

```
{
  _id: ObjectId("4d6f29c0e4ef0123afdacaeb"),
  filename: "sheet-1",
  updated_at: Time.now.utc,
  username: "banks",
  data => "RAW DATA"
}
```

Note that the `data` field will contain a 5KB string to simulate the raw data.

I've written a little Ruby script that we can use to simulate writes to the cluster. At each iteration, the script inserts one 5 KB document for each of two-hundred users. The program is included in the source distribution, but you can read the most important part of the code below. Notice what we connect directly to the *mongos* process on port 40000 and that we use the standard `Mongo::Connection` class (and not the `Mongo::ReplSetConnection` class). This is because the *mongos* process contains all the logic needed for connecting to the individual replica sets, *qua* shards, and for handling failover.

```
require 'rubygems'
require 'mongo'
require 'names'

#<start id="write_docs">
@con = Mongo::Connection.new("localhost", 40000)
@col = @con['cloud-docs']['spreadsheets']
@data = "abcde" * 1000

def write_user_docs(iterations=0, name_count=200)
  iterations.times do |n|
    name_count.times do |n|
      doc = { :filename => "sheet-#{n}",
              :updated_at => Time.now.utc,
              :username => Names::LIST[n],
              :data => @data
            }
      @col.insert(doc)
    end
  end
end
#<start id="write_docs">

if ARGV.empty? || !(ARGV[0] =~ /^d+$/)
```

```
  puts "Usage: load.rb [iterations] [name_count]"
else
  iterations = ARGV[0].to_i

  if ARGV[1] && ARGV[1] =~ /^d+$/
    name_count = ARGV[1].to_i
  else
    name_count = 200 end

    write_user_docs(iterations, name_count)
end
```

If you have the program on hand, go ahead and run it.

```
$ ruby load.rb
```

Now connect to the *mongos* using the shell. If you query the *spreadsheets* collection, you'll see that it contains exactly two-hundred documents and that they total around 1 MB. You can also query a document, but be sure to exclude the sample data field (since you don't want to print 5 KB of text to the screen).

```
$ mongo localhost:40000
> use cloud-docs
> db.spreadsheets.count()
200
> db.spreadsheets.stats()['size']
1019496
> db.spreadsheets.findOne({}, {data: 0})
{
  "_id" : ObjectId("4d6d6b191d41c8547d0024c2"),
  "username" : "Wagner",
  "updated_at" : ISODate("2011-03-01T21:54:33.813Z"),
  "filename" : "sheet-9"
}
```

Now we can check out what's happened sharding wise. Switch to the config database and check the number of chunks.

```
> use config
> db.chunks.count()
1
```

There's just one chunk so far. Let's see how it looks.

```
> db.chunks.findOne()
{
  "_id" : "cloud-docs.spreadsheets-username_MinKey_id_MinKey",
  "lastmod" : {
    "t" : 1000,
    "i" : 0
  },
  "ns" : "cloud-docs.spreadsheets",
  "min" : {
    "username" : { $minKey : 1 },
    "_id" : { $minKey : 1 }
  },
  "max" : {
    "username" : { $maxKey : 1 },
    "_id" : { $maxKey : 1 }
  },
  "shard" : "shard-a"
}
```

Can you figure out what range this chunk represents? If there's just one chunk, then it spans the entire sharded collection. That's borne out by the `min` and `max`

> **MIN KEY AND MAX KEY** `$minKey` and `$maxKey` are proper BSON types that, respectively, compare lower than and greater than all other BSON types. Because the value for any given field can contain any BSON type, we need these two types to mark the chunks endpoints at the extremities of the sharded collection.

We can see a more interesting chunk range by adding more data to our *spreadsheets* collection. The load script takes an optional parameter specifying the number of insertions to make. We'll do 100, which will insert an extra 20,000 documents totaling 100MB.

```
$ ruby load.rb 100
```

Verify that the insert worked:

```
> db.spreadsheets.count()
20200
> db.spreadsheets.stats()['size']
102989600
```

Having inserted this much data, we'll definitely have more than one chunk. Let's check the chunk state with the `db.printShardingSizes()` method.

```
> use config
> db.chunks.count()
10

> db.printShardingSizes()
--- Sharding Status ---
  sharding version: { "_id" : 1, "version" : 3 }

  shards:
      { "_id" : "shard-a", "host" : "shard-a/localhost:30000,localhost:30001" }
      { "_id" : "shard-b", "host" : "shard-b/localhost:30100,localhost:30101" }

  databases:
  { "_id" : "admin", "partitioned" : false, "primary" : "config" }
  { "_id" : "test", "partitioned" : false, "primary" : "shard-b" }
  { "_id" : "cloud-docs", "partitioned" : true, "primary" : "shard-a" }

    cloud-docs.spreadsheets chunks:
      { "username" : { $minKey : 1 }, "_id" : { $minKey : 1 } } -->>
      { "username" : "Abbott", "_id" : ObjectId("4d6d57f61d41c851ee000092") }
        on : shard-b { "estimate" : false, "size" : 0, "numObjects" : 0 }

      { "username" : "Abbott", "_id" : ObjectId("4d6d57f61d41c851ee000092") } -->>
      { "username" : "Chen", "_id" : ObjectId("4d6d59db1d41c8536f001453") }
        on : shard-b { "estimate" : false, "size" : 17130516, "numObjects" : 3360 }

      { "username" : "Chen", "_id" : ObjectId("4d6d59db1d41c8536f001453") } -->>
      { "username" : "Finch", "_id" : ObjectId("4d6d59df1d41c8536f003f18") }
        on : shard-b { "estimate" : false, "size" : 10058360, "numObjects" : 1973 }

      { "username" : "Finch", "_id" : ObjectId("4d6d59df1d41c8536f003f18") } -->>
      { "username" : "Hester", "_id" : ObjectId("4d6d59db1d41c8536f000f54") }
        on : shard-b { "estimate" : false, "size" : 12562332, "numObjects" : 2464 }

      { "username" : "Hester", "_id" : ObjectId("4d6d59db1d41c8536f000f54") } -->>
      { "username" : "Lawrence", "_id" : ObjectId("4d6d59db1d41c8536f000fb2") }
        on : shard-b { "estimate" : false, "size" : 12362792, "numObjects" : 2425 }

      { "username" : "Lawrence", "_id" : ObjectId("4d6d59db1d41c8536f000fb2") } -->>
      { "username" : "Nichols", "_id" : ObjectId("4d6d59d81d41c8536f00017e") }
        on : shard-a { "estimate" : false, "size" : 14322316, "numObjects" : 2809 }

      { "username" : "Nichols", "_id" : ObjectId("4d6d59d81d41c8536f00017e") } -->>
      { "username" : "Sharpe", "_id" : ObjectId("4d6d59db1d41c8536f00154e") }
        on : shard-a { "estimate" : false, "size" : 15067856, "numObjects" : 2955 }

      { "username" : "Sharpe", "_id" : ObjectId("4d6d59db1d41c8536f00154e") } -->>
      { "username" : "Underwood", "_id" : ObjectId("4d6d59de1d41c8536f00354f") }
        on : shard-a { "estimate" : false, "size" : 9993628, "numObjects" : 1960 }

      { "username" : "Underwood", "_id" : ObjectId("4d6d59de1d41c8536f00354f") } -->>
      { "username" : "Zhang", "_id" : ObjectId("4d6d59da1d41c8536f000cf6") }
        on : shard-a { "estimate" : false, "size" : 11063736, "numObjects" : 2170 }

      { "username" : "Zhang", "_id" : ObjectId("4d6d59da1d41c8536f000cf6") } -->>
      { "username" : { $maxKey : 1 }, "_id" : { $maxKey : 1 } }
        on : shard-a { "estimate" : false, "size" : 428064, "numObjects" : 84 }
```

The picture has definitely changed. We now have ten chunks, each of which averages around 10 MB in size. Naturally, each chunk represents a contiguous range of data. You can see that the first chunk with data comprises documents from "Abbott" to "Chen" and the last chunks runs from "Zhang" to the $maxKey. But not only do we

have more chunks; the chunks have actually migrated to shard B. You could scan the foregoing list for this, but here's a really quick way to check the chuck distribution:

```
> db.chunks.count({"shard": "shard-a"})
5
> db.chunks.count({"shard": "shard-b"})
5
```

As long as the cluster's data size is small, the splitting algorithm dictates that splits happen often. That's what we see now. This gives us a good distribution of data and chunks early on. If writes remain roughly evenly distributed across the chunk range, the few migrates will occur.

> **EARLY CHUNK SPLITTING** A sharded cluster will split chunks aggressively early on to expedite the migration of data across shards. Specifically, when the number of chunks is less than ten, chunks will split at one quarter of the max chunk size (16 MB), and when the number of chunks is between ten and twenty, they'll split at half the maximum chunk size (32 MB). This has two nice benefits. First, it creates a lot of chunks up front, which initiates a migration round. And, second, that migration round occurs fairly painlessly, as the small chunk size ensures that the total amount of data migrated is small.

Now the split threshold will increase. We can see how the splitting slows down, and how chunks start to grow toward their max size, by doing a much more massive insert. Here we add another 800 MB to the cluster.

```
ruby load.rb 800
```

Note that we just increased the total data size by a factor of eight. But if you check the chunking status, you'll see that there are only twice as many chunks.

```
> use config
> db.chunks.count()
21
```

We have more chunks, which means that the chunk ranges will be smaller, but each chunk is being allowed to grow much larger. So, for example, the first chunk in our collection spans from "Abbott" to "Bender" but is already nearly 60 MB large.

```
> use config
> db.chunks.findOne()
{ "username" : "Abbott", "_id" : ObjectId("4d6d57f61d41c851ee000092") } -->>
{ "username" : "Bender", "_id" : ObjectId("4d6d696e1d41c8543e000009") }
  on : shard-b { "estimate" : false, "size" : 59718984, "numObjects" : 11713 }
```

Because the max chunk size is currently 64 MB, we'd soon see this chunk split if we were to continue inserting data. Notice too that the distribution still looks pretty much even, as it was before.

```
> db.chunks.count({"shard": "shard-a"})
11
> db.chunks.count({"shard": "shard-b"})
10
```

Although the number of chunks has increased during the last 800 MB insert round, we can probably assume that not a single migrate occurred; a likely scenario is that each of the original chunks split in two, with a single extra split somewhere in the mix. We can verify this by querying the config database's `changelog` collection:

```
> db.changelog.count({what: "split"})
20
> db.changelog.find({what: "moveChunk.commit"}).count()
6
```

This is certainly in line with our assumptions. Twenty splits have occurred, yielding twenty-one chunks, but only six migrates have taken place. For an extra deep look at what's going on here, we can scan the change log entries. For instance, here's the entry recording the first chunk move.

```
> db.changelog.findOne({what: "moveChunk.commit"})
{
  "_id" : "ubuntu-2011-03-01T20:40:59-2",
  "server" : "ubuntu",
  "clientAddr" : "127.0.0.1:55749",
  "time" : ISODate("2011-03-01T20:40:59.035Z"),
  "what" : "moveChunk.commit",
  "ns" : "cloud-docs.spreadsheets",
  "details" : {
    "min" : {
      "username" : { $minKey : 1 },
      "_id" : { $minKey : 1 }
```

```
        },
        "max" : {
          "username" : "Abbott",
          "_id" : ObjectId("4d6d57f61d41c851ee000092")
        },
        "from" : "shard-a",
        "to" : "shard-b"
      }
    }
```

Here we see the movement of a chunk from "shard-a" to "shard-b." In general, the documents in the change log are quite readable.

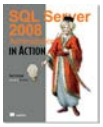
Summary

Sharding is an effective strategy for maintaining high read and write performance on large data sets. MongoDB's sharding works well in numerous production deployments and can work for you, too. Instead of having to worry about implementing your own half-baked, custom sharding solution, you can take advantage of all the effort that's been put into MongoDB's sharding mechanism.

Here are some other Manning titles you might be interested in:

SQL Server MVP Deep Dives
Paul Nielsen, Kalen Delaney, Greg Low, Adam Machanic, Paul S. Randal, and Kimberly L. Tripp

SQL Server DMVs in Action
Ian W. Stirk



SQL Server 2008 Administration in Action
Rod Colledge

Last updated: January 20, 2012