CHAPTER 12

Sharding

Whether you're building the next Facebook or just a simple database application, you will probably need to scale your app up at some point if it's successful. If you don't want to be continually replacing your hardware, then you will want to use a technique that allows you to add capacity incrementally to your system, as you need it. *Sharding* is a technique that allows you to spread your data across multiple machines, yet does so in a way that mimics an app hitting a single database.

Ideally suited for the new generation of cloud-based computing platforms, sharding as implemented by MongoDB is perfect for dynamic, load-sensitive automatic scaling, where you ramp up your capacity as you need it and turn it down when you don't.

This chapter will walk you through implementing sharding in MongoDB.

Exploring the Need for Sharding

When the World Wide Web was just getting under way, the number of sites, users, and the amount of information available online was low. The Web consisted of a few thousand sites and a population of only tens or perhaps hundreds of thousands of users predominantly centered on the academic and research communities. In those early days, data tended to be simple: hand-maintained HTML documents connected together by hyperlinks. The original design objective of the protocols that make up the Web was to provide a means of creating navigable references to documents stored on different servers around the Internet.

Even current big brand names such as Yahoo had only a minuscule presence on the Web compared to its offerings today. The Yahoo directory that comprised the original product around which the company was formed was little more than a network of hand-edited links to popular sites. These links were maintained by a small but enthusiastic band of people called *the surfers*. Each page in the Yahoo directory was a simple HTML document stored in a tree of filesystem directories and maintained using a simple text editor.

But as the size of the net started to explode—and the number of sites and visitors started its near-vertical climb upwards—the sheer volume of resources available forced the early Web pioneers to move away from simple documents to more complex dynamic page generation from separate data stores.

Search engines started to spider the Web and pull together databases of links that today number in the hundreds of billions of links and tens of billions of stored pages.

These developments prompted the movement to datasets managed and maintained by evolving content management systems that were stored mainly in databases for easier access.

At the same time, new kinds of services evolved that stored more than just documents and link sets. For example, audio, video, events, and all kinds of other data started to make its way into these huge datastores. This is often described as the "industrialization of data"—and in many ways it shares parallels with the evolution of the industrial revolution centered on manufacturing during the 19th century.

Eventually, every successful company on the Web faces the problem of how to access the data stored in these mammoth databases. They find that there are only so many queries per second that can be handled with a single database server, and network interfaces and disk drives can only transfer so many megabytes per second to and from the web servers. Companies that provide web-based services can quickly find themselves exceeding the performance of a single server, network, or drive array. In such cases, they are compelled to divide and distribute their massive collections of data. The usual solution is to *partition* these mammoth chunks of data into smaller pieces that can be managed more reliably and quickly. At the same time, these companies need to maintain the ability to perform operations across the entire breadth of the data held in their large clusters of machines.

Replication, which you learned about in some detail in Chapter 11, can be an effective tool for overcoming some of these scaling issues, enabling you to create multiple copies of your data in multiple servers. This enables you to spread out your server load across more machines.

Before long, however, you run headlong into another problem, where the size of the individual tables or collections that make up your data set grow so large that they exceed the capacity of a single database system to manage them effectively. For example, Flickr announced that on October 12th 2009 it had received its 4 *billionth* photo, and the site is now well on its way to crossing the 10 billion photos mark.

Attempting to store the details of 10 billion photos in one table is not feasible, so Flickr looked at ways of distributing that set of records across a large number of database servers. The solution adopted by Flickr serves as one of the better-documented (and publicized) implementations of sharding in the real world.

Partitioning Horizontal and Vertical Data

Data partitioning is the mechanism of splitting data across multiple independent datastores. Those datastores can be co-resident (on the same system) or remote (on separate systems). The motivation for co-resident partitioning is to reduce the size of individual indices and reduce the amount of I/O that is needed to update records. The motivation for remote partitioning is to increase the bandwidth of access to data, by having more network interfaces and disc data I/O channels available.

Partitioning Data Vertically

In the traditional view of databases, data is stored in rows and columns. Vertical partitioning consists of breaking up a record on column boundaries and storing the parts in separate tables or collections. It can be argued that a relational database design that uses joined tables with a one-to-one relationship is a form of co-resident vertical data partitioning.

MongoDB, however, does not lend itself to this form of partitioning because the structure of its records (documents) does not fit the nice and tidy row and column model. Therefore, there are few opportunities to cleanly separate a row based on its column boundaries. MongoDB also promotes the use of *embedded* documents, and it does not directly support the ability to *join* associated collections together.

Partitioning Data Horizontally

Horizontal partitioning is where all the action is when using MongoDB, and *sharding* is the common term for a popular form of horizontal partitioning. Sharding allows you to split a collection across multiple servers to improve performance in a collection that has a large number of documents in it.

A simple example of sharding occurs when a collection of user records is divided across a set of servers, so that all the records for people with last names that begin with the letters A–G are on one server, H–M are on another, and so on. The rule that splits the data is known as the *sharding key function*, or the *data hashing function*.

In simple terms, sharding allows you to treat the *cloud* of shards as through it were a single collection, and an application does not need to be aware that the data is distributed across multiple machines. Traditional sharding implementations require the application to be actively involved in determining which server a particular document is stored on, so it can route its requests properly. Traditionally, there is a library bound to the application, and this library is responsible for storing and querying data in sharded data sets.

MongoDB is virtually unique in its support for *auto-sharding*, where the database server manages the splitting of the data and the routing of requests to the required shard server. If a query requires data from multiple *shards*, then MongoDB will manage the process of merging the data obtained from each shard back into a single cursor.

This feature, more than any other, is what earns MongoDB its stripes as a *cloud* or web-oriented database.

Analyzing a Simple Sharding Scenario

Let's assume you want to implement a simple sharding solution for a fictitious Gaelic social network. Figure 12–1 shows a simplified representation of how this application could be sharded.

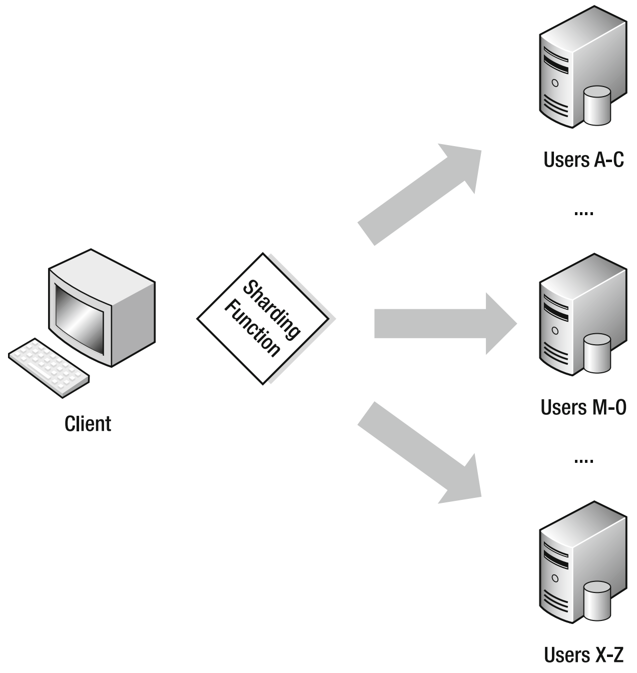

Figure 12–1. *Simple sharding of a User collection*

There are a number of problems with this simplified view of our application. Let's look at the most obvious ones.

First, if your Gaelic network is targeted at the Irish and Scottish communities around the world, then the database will have a large number of names that start with *Mac* and *Mc* (e.g., MacDonald, McDougal, and so on) for the Scottish population and *O'* (e.g., O'Reilly, O'Conner, and so on) for the Irish population. Thus, using the simple sharding key function based on the first letter of the last name will place an undue number of user records on the shard that supports the letter range "M–O." Similarly, the shard that supports the letter range "X–Z" will perform very little work at all.

An important characteristic of a sharding system is that it must ensure that the data is spread evenly across the available set of shard servers. This prevents *hotspots* from developing that can affect the overall performance of the cluster. Let's call this *Requirement 1: The ability to distribute data evenly across all shards.*

Another thing to keep in mind: when you split your dataset across multiple servers, you effectively increase your dataset's vulnerability to hardware failure. That is, you increase the chance that a single server failure will affect the availability of your data as you add servers. Again, an important characteristic of a reliable sharding system is that—like a RAID system commonly used with disk drives—it stores each piece of data on more than one server, and it can tolerate individual shard servers becoming unavailable. Let's call this *Requirement 2: The ability to store shard data in a fault-tolerant fashion*.

Finally, you want to make sure that you can add or remove servers from the set of shards without having to back up and restore the data and *redistribute* it across a smaller or larger set of shards. Further, you need to be able to do this without causing any downtime on the cluster. Let's call this *Requirement 3: The ability to add or remove shards while the system is running*.

The upcoming sections will cover how to address these requirements.

Implementing Sharding with MongoDB

MongoDB uses a *proxy* mechanism to support sharding (see Figure 12–2); the provided mongos daemon acts as a *controller* for multiple *mongod*-based shard servers. Your application attaches to the mongos daemon as though it were a single MongoDB database server; thereafter, your application sends all of its commands (e.g., updates, queries, and deletes) to that mongos daemon.

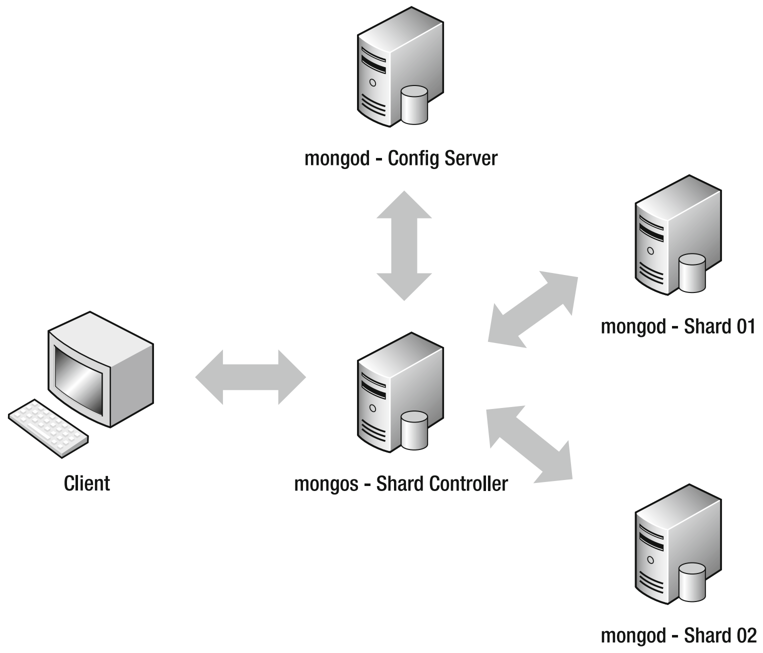

Figure 12–2. *A simple sharding setup without redundancy*

The mongos daemon is responsible for managing which MongoDB server is sent the commands from your application, and this daemon will reissue queries that cross multiple shards to multiple servers and aggregate the results together.

MongoDB implements sharding at the collection level, not the database level. In many systems, only one or two collections may grow to the point where sharding is required. Thus, sharding should be used judiciously; you don't want to impose the overhead of managing the distribution of data for smaller collections if you don't need to.

Let's return to the fictitious Gaelic social network example. In this application, the `user` collection contains details about its users and their profiles. This collection is likely to grow to the point where it needs to be sharded. However, other collections such as `events`, `countries`, and `states` are unlikely to ever become so large that sharding would provide any benefit.

The sharding system uses a sharding key function to map data into *chunks*, which are blocks of storage containing documents (see Chapter 5 for more information on chunks). Each chunk stores documents with a particular continuous range of sharding key values; these values enable the mongos controller to quickly find a chunk that contains a document it needs to work on. MongoDB's sharding system then stores this chunk on an available shard store; the config servers keep track of which chunk is stored on which shard server. This is an important feature of the implementation because it allows you to add and remove shards from a cluster without having to back up and restore the data.

When you add a new shard to the cluster, the system will redistribute its chunks across the new set of servers in order to improve performance. Similarly, when you remove a shard, the sharding controller will *drain* the chunks out of the shard being taken offline and redistribute them to the remaining shard servers.

A sharding setup for MongoDB also needs a place to store the configuration of its shards, as well as a place to store information about each shard server in the cluster. To support this, a MongoDB server called a *config server* is required; this server instance is a normal mongod server running in a special role. As explained above, the config servers also act as directories that allow the location of each chunk to be determined

At first glance, it appears that implementing a solution that relies on sharding requires a lot of servers! However, similar to what you saw in Chapter 11's coverage of replication, you can co-host multiple instances of each of the different services required to create a sharding setup on a relatively small number of physical servers. Figure 12–3 shows a fully redundant sharding system that uses replica sets for the shard storage and the config servers, as well as a set of mongos daemons to manage the cluster. It also shows how those services can be condensed to run on just three physical servers.

Carefully placing the shard storage instances so that they are correctly distributed among the physical servers enables you to ensure that your system can tolerate the failure of one or more servers in your cluster. This mirrors the approach used by RAID disk controllers to distribute data across multiple drives in stripes, enabling RAID configurations to recover from a failed drive.

■ **Note** If you don't supply a sharding key, then MongoDB will automatically use the `_id` field to distribute documents among your database's shards; however, this field may not provide the optimal data value to use to shard your data.

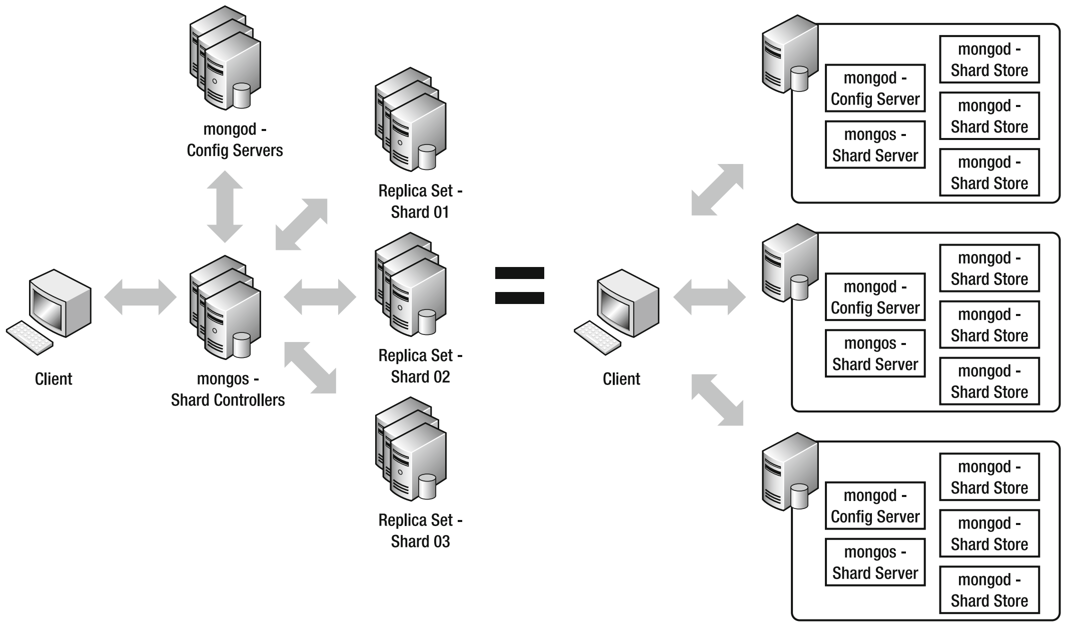

Figure 12–3. *A redundant sharding configuration*

Setting Up a Sharding Configuration

To use sharding effectively, it's important that you understand how it works. The next example will walk you through how to set up a test configuration on a single machine. You will configure this example like the simple sharding system shown in Figure 12–2, with one difference: this example will keep things simple by using only two shards. Finally, you will learn how to create a sharded collection and a simple PHP test program that demonstrates how to use this collection.

In this test configuration, you will use the services listed in Table 12–1.

Table 12–1. *Server Instances in the Test Configuration*

Service	Daemon	Port	Dbpath
Shard Controller	mongos	27021	N/A
Config Server	mongod	27022	`/db/config/data`
Shard0	mongod	27023	`/db/shard1/data`
Shard1	mongod	27024	`/db/shard2/data`

Let's begin by setting up the configuration server. Do so by opening a new terminal window and typing the following code.

```
$sudo mkdir -p /db/config/data
$sudo mongod --port 27022 --dbpath /db/config/data --configsvr
```

Be sure to leave your terminal window open once you have the config server up and running. Next, you need to set up the shard controller (mongos). To do so, open a new terminal window and type the following:

```
$sudo mongos --configdb localhost:27022 --port 27021 --chunkSize 1
```

This brings up the shard controller, which should announce that it's listening on port 27021. If you look at the terminal window for the config server, you should see that the shard server has connected to its config server and registered itself with it.

In this example, you set the chunk size to its smallest possible size of 1 MB. This is not a practical value for real-world systems because it means that the chunk storage is smaller than the maximum size of a document (4 MB). However, this is just a demonstration, and the small chunk size allows you to create a lot of chunks to exercise the sharding setup without also having to load a lot of data. By default, `chunkSize` is set to 128 MB unless otherwise specified.

Finally, you're ready to bring up the two shard servers. To do so, you will need two fresh terminal windows, one for each server. Type the following into one window to bring up the first server:

```
$sudo mkdir -p /db/shard0/data
$sudo mongod --port 27023 --dbpath /db/shard0/data --shardsvr
```

And type the following into the second window to bring up the second server:

```
$sudo mkdir -p /db/shard1/data
$sudo mongod --port 27024 --dbpath /db/shard1/data --shardsvr
```

You have your servers up and running. Next, you need to tell the sharding system where the shard servers are located. To do this, you need to connect to your shard controller (mongos). It's important to remember that, even though mongos is not a full MongoDB instance, it appears to be a full instance to your application. Therefore, you can just use the mongo command shell to attach to the shard controller and add your two shards, as shown in the following example:

```
$ mongo localhost:27021
> use admin
switched to db admin
> db.runCommand( { addshard : "localhost:27023", allowLocal : true } )
{ "added" : "localhost:27023", "ok" : 1 }
> db.runCommand( { addshard : "localhost:27024", allowLocal : true } )
{ "added" : "localhost:27024", "ok" : 1 }
```

Your two shard servers are now activated; next, you need to check the shards using the `listshards` command:

```
> db.runCommand({listshards:1})
{
    "shards" : [
        {
            "_id" : "shard0",
            "host" : "localhost:27023"
        },
        {
            "_id" : "shard1",
            "host" : "localhost:27024"
        }
    ],
    "ok" : 1
}
```

You now have a working sharded server; next, you will create a new database called `testdb`, and then activate a collection called `testcollection` inside this database. You will shard this collection, so you will give this collection an entry called `testkey` that you will use as the sharding function:

```
> testdb = db.getSisterDB("testdb")
testdb
> db.runCommand({ enablesharding: "testdb"})
{ "ok" : 1 }
> db.runCommand({ shardcollection : "testdb.testcollection", key : {testkey : 1}})
{ "collectionsharded" : "testdb.testcollection", "ok" : 1 }
```

Thus far, you have created a sharded cluster with two shard storage servers. You have also created a database on it with a sharded collection. A server without any data in it is of no use to anybody, so it's time to get some data into this collection, so you can see how the shards are distributed.

To do this, you will use a small PHP program to load the sharded collection with some data. The data you will load consists of a single field called `testkey`. This field contains a random number and a second field with a fixed chunk of text inside it (the purpose of this second field is to make sure you can create a reasonable number of chunks to shard). This collection serves as the main data table for a fictitious website called `TextAndARandomNumber.com`. The following code creates a PHP program that inserts data into your sharded server:

```
<?php
// Open a database connection to the mongos daemon
$mongo = new Mongo("localhost:27021");
// Select the test database
$db = $mongo->selectDB('testdb');
// Select the TestIndex collection
$collection = $db->testcollection;

for($i=0; $i < 100000 ; $i++){
        $data=array();
        $data['testkey'] = rand(1,100000);
        $data['testtext'] = "Because of the nature of MongoDB, many of the more "
                            . "traditional functions that a DB Administrator "
                            . "would perform are not required.  Creating new databases, "
                            . "collections and new fields on the server are no longer
necessary, "
                            . "as MongoDB will create these elements on-the-fly as you access
them."
                            . "Therefore, for the vast majority of cases managing databases
and "
                            . "schemas is not required.";
        $collection->insert($data);
}
```

This small program will connect to the shard controller (mongos) and insert 100,000 records with random testkeys and some **`testtext`** to pad out the documents. As mentioned previously, this sample text causes these documents to occupy a sufficient number of chunks to make using the Sharding mechanism feasible.

The following command runs the test program:

```
$php testshard.php
```

Once the program has finished running, you can connect to the mongos instance with the command shell and verify that the data has been stored:

```
$mongo localhost:27021
>use testdb
>db.testcollection.count()
100000
```

At this point, you can see that your server has stored 100,000 records. Now you need to connect to each shard and see how many items have been stored in `testdb.testcollection` for each shard. The following code enables you to connect to the first shard and see how many records are stored in it from the `testcollection` collection:

```
$mongo localhost:27023
>use testdb
>db.testcollection.count()
48875
```

And this code enables you to connect to the second shard and see how many records are stored in it from the `testcollection` collection:

```
$mongo localhost:27024
>use testdb
>db.testcollection.count()
51125
```

■ **Note** You may see different values for the number of documents in each shard, depending on when exactly you look at the individual shards. The mongos instance may initially place all the chunks on one shard, but over time it will *rebalance* the shard set to evenly distribute data among all the shards by moving chunks around. Thus, the number of records stored in a given shard may change from moment to moment. This satisfies “Requirement 1: The ability to distribute data evenly across all shards.”

Adding a New Shard to the Cluster

Let’s assume business is really jumping at `TextAndARandomNumber.com`. To keep up with the demand, you decide to add a new shard server to the cluster to spread out the load a little more.

Adding a new shard is easy; all it requires is that you repeat the steps described previously. Begin by creating the new shard storage server and place it on port 27025, so it does not clash with your existing servers:

```
$ sudo mkdir -p /db/shard2/data
$ sudo mongod --port 27025 --dbpath /db/shard2/data --shardsvr
```

Next, you need to add the new shard server to the cluster. You do this by logging into the sharding controller (mongos), and then using the admin `addshard` command:

```
$mongo localhost:27021
>use admin
admin
>db.runCommand( { addshard : "localhost:27025", allowLocal : true } )
```

At this point, you can run the `listshards` command to verify that the shard has been added to the cluster. Doing so reveals that a new shard server (shard2) is now present in the `shards` array:

```
>db.runCommand({listshards:1})
{
    "shards" : [
{
            "_id" : "shard0",
            "host" : "localhost:27023"
        },
        {
            "_id" : "shard1",
            "host" : "localhost:27024"
        },
        {
            "_id" : "shard2",
            "host" : "localhost:27025"
        }

    ],
    "ok" : 1
}
```

If you log in to the new shard storage server you have created on port 27025 and look at `testcollection`, you will see something interesting:

```
$mongo localhost:27025
> use testdb
switched to db testdb
> show collections
system.indexes
testcollection
> db.testcollection.count()
4657
> db.testcollection.count()
4758
> db.testcollection.count()
6268
```

This shows that the number of items in the `testcollection` on your new shard2 storage server is slowly going up. What you are seeing is proof that the sharding system is rebalancing the data across the expanded cluster. Over time, the sharding system will migrate chunks from the shard0 and shard1 storage servers to create an even distribution of data across the three servers that make up the cluster. This process is automatic, and it will happen even if there is no new data being inserted into the `testcollection` collection. In this case, the mongos shard controller is moving chunks to the new server, and then registering them with the config server.

This is one of the factors to consider when choosing a chunk size. If your chunkSize value is very large, then you will get a less even distribution of data across your shards; conversely, the smaller your chunkSize value, the more even the distribution of your data will be.

Removing a Shard from the Cluster

It was great while it lasted, but now assume that TextAndARandomNumber.com was a flash in the pan and its sizzle fizzled. After a few weeks of frenzied activity, the site's traffic started to fall off, so you had to start to look for ways to cut your running costs—in other words, that new shard server had to go!

In the next example, you will remove the shard server you added previously. To initiate this process, log in to the shard controller (mongos) and issue the removeShard command:

```
$ mongo localhost:27021
> use admin
switched to db admin
> db.runCommand({removeShard : "localhost:27025"})
{
    "msg" : "draining started successfully",
    "state" : "started",
    "shard" : "shard2",
    "ok" : 1
}
```

The removeShard command responds with a message indicating that the removal process has started. It also indicates that the shard controller (mongos) has begun relocating the chunks on the target shard server to the other shard servers in the cluster. This process is known as *draining* the shard.

You can check the progress of the draining process by reissuing the removeShard command. The response will tell you how many chunks and databases still need to be drained from the shard:

```
> db.runCommand({removeShard : "localhost:27025"})
{
    "msg" : "draining ongoing",
    "state" : "ongoing",
    "remaining" : {
        "chunks" : NumberLong( 12 ),
        "dbs" : NumberLong( 0 )
    },
    "ok" : 1
}
```

Finally, the removeShard process will terminate, and you will get a message indicating that the removal process is complete:

```
> db.runCommand({removeShard : "localhost:27025"})
{
    "msg" : "removeshard completed successfully",
    "state" : "completed",
    "shard" : "shard2",
    "ok" : 1
}
```

To verify that the removeShard command was successful, you can run listshards to confirm that the desired shard server has been removed from the cluster. For example, the following output shows that the shard2 server that you created previously is no longer listed in the shards array:

```
>db.runCommand({listshards:1})
{
    "shards" : [
        {
            "_id" : "shard0",
            "host" : "localhost:27023"
        },
        {
            "_id" : "shard1",
            "host" : "localhost:27024"
        }
    ],
    "ok" : 1
}
```

At this point, you can terminate the Shard2 mongod process and delete its storage files because its data has been migrated back to the other servers.

■ **Note** The ability to add and remove shards to your cluster without having to take it offline is a critical component of MongoDB's ability to support highly scalable, highly available, large-capacity datastores. This satisfies the final requirement: "Requirement 3: The ability to add or remove shards while the system is running."

Determining How You're Connected

Your application can be connected either to a standard non-sharded database (mongod) or to a shard controller (mongos). MongoDB makes both of these processes; for all but a few use cases, the database and shard controller look and behave exactly the same way. However, sometimes it may be important to determine what type of system you are connected to.

MongoDB provides the `isdbgrid` command, which you can use to interrogate the connected data system to determine whether it is sharded. The following snippet shows how to use this command, as well as what its output looks like:

```
$mongo
>use testdb
>db.runCommand({ isdbgrid : 1});
{ "isdbgrid" : 1, "hostname" : "localhost", "ok" : 1 }
```

The response includes the `isdbgrid:1` field, which tells you that the database you are connected to is enabled for sharding. A response of `isdbgrid:0` would indicate that you are connected to a non-sharded database.

Listing the Status of a Sharded Cluster

MongoDB also includes a simple command for dumping the status of a sharding cluster: `printShardingStatus()`.

This command can give you a lot of insight into the internals of the sharding system. The following snippet shows how to invoke the `printShardingStatus()` command, but strips out some of the output returned to make it easier to read:

```
 $mongo localhost:27021
>use admin
>db.printShardingStatus();
--- Sharding Status ---
  sharding version: { "_id" : 1, "version" : 3 }
  shards:
      { "_id" : "shard0", "host" : "localhost:27023" }
      { "_id" : "shard1", "host" : "localhost:27024" }
  databases:
    { "_id" : "admin", "partitioned" : false, "primary" : "config" }
    { "_id" : "testdb", "partitioned" : true, "primary" : "shard0",
       "sharded" : { "testdb.testcollection" : { "key" : { "testkey" : 1 }, "unique" : false }
} }
        testdb.testcollection chunks:
            { "testkey" : { $minKey : 1 } } -->> { "testkey" : 47 } on : shard1 { "t" : 2000,
"i" : 2 }
            { "testkey" : 47 } -->> { "testkey" : 1702 } on : shard1 { "t" : 66000, "i" : 0 }
            { "testkey" : 1702 } -->> { "testkey" : 3356 } on : shard1 { "t" : 66000, "i" : 1
}
            { "testkey" : 3356 } -->> { "testkey" : 5020 } on : shard1 { "t" : 66000, "i" : 2
}
            { "testkey" : 5020 } -->> { "testkey" : 6679 } on : shard1 { "t" : 66000, "i" : 3
}
            { "testkey" : 6679 } -->> { "testkey" : 8137 } on : shard1 { "t" : 66000, "i" : 4
}
            { "testkey" : 8137 } -->> { "testkey" : 9637 } on : shard1 { "t" : 66000, "i" : 5
}

...

            { "testkey" : 64878 } -->> { "testkey" : 66399 } on : shard0 { "t" : 66000, "i" :
152 }
            { "testkey" : 66399 } -->> { "testkey" : 67934 } on : shard0 { "t" : 66000, "i" :
148 }
            { "testkey" : 67934 } -->> { "testkey" : 69410 } on : shard0 { "t" : 66000, "i" :
43 }
            { "testkey" : 69410 } -->> { "testkey" : 70962 } on : shard0 { "t" : 66000, "i" :
44 }
            { "testkey" : 70962 } -->> { "testkey" : 72470 } on : shard0 { "t" : 66000, "i" :
45 }
            { "testkey" : 72470 } -->> { "testkey" : 73991 } on : shard0 { "t" : 66000, "i" :
46 }
            { "testkey" : 73991 } -->> { "testkey" : 75478 } on : shard0 { "t" : 66000, "i" :
47 }
            { "testkey" : 75478 } -->> { "testkey" : 77063 } on : shard0 { "t" : 66000, "i" :
48 }
            { "testkey" : 77063 } -->> { "testkey" : 78654 } on : shard0 { "t" : 66000, "i" :
49 }

...
```

```
            { "testkey" : 96954 } -->> { "testkey" : 98468 } on : shard0 { "t" : 66000, "i" :
62 }
            { "testkey" : 98468 } -->> { "testkey" : 99979 } on : shard0 { "t" : 66000, "i" :
145 }
            { "testkey" : 99979 } -->> { "testkey" : { $maxKey : 1 } } on : shard0 { "t" :
6000, "i" : 4 }

>
```

This output lists the shard servers, the configuration of each sharded database/collection, and each chunk in the sharded dataset. Because you used a small `chunkSize` value to simulate a larger sharding setup, this report lists a lot of chunks. An important piece of information that can be obtained from this listing is the range of *sharding keys* associated with each chunk. The preceding output also shows which shard server the specific chunks are stored on. You can use the output returned by this command as the basis for a tool to analyze the distribution of a shard server's keys and chunks. For example, you might use this data to determine whether there is any *clumping* of data in the dataset.

Using Replica Sets to Implement Shards

The examples you have seen so far rely on a single mongod instance to implement each shard. In Chapter 11, you learned how to create replica sets, which are clusters of mongod instances working together to provide redundant and fail-safe storage.

When adding shards to the sharded cluster, you can provide the name of a replica set and the address of a member of that replica set, and that shard will be instanced on each of the replica set members. Mongos will track which instance is the primary server for the replica set; it will also make sure that all shard writes are made to that instance.

Combining sharding and replica sets enables you to create high-performance, highly reliable clusters that can tolerate multi-machine failure. It also enables you to maximize the performance and availability of cheap, commodity-class hardware.

■ **Note** The ability to use replica sets as a storage mechanism for shards satisfies "Requirement 2: The ability to store shard data in a fault-tolerant fashion."

Sharding to Improve Performance

In the MongoDB versions available at the time of writing, several MongoDB operations are *single-threaded* only, such as background indexing and aggregate functions (e.g., grouping and map/reduce). The JavaScript engine (spidermonkey) used in MongoDB also imposes this restriction.

Modern CPUs support features such as multiple cores and hyperthreading. These features enable simultaneous code execution, so it is advantageous to break up long-running tasks into separate processes to take advantage of the multiple cores.

If you have a four core CPU, it may be advantageous to create a four-shard database configuration inside a single physical machine because all of the aforementioned operations will be passed to the individual shard storage server instances for execution. Doing this enables each core to handle a separate operation at the same time.

■ **Caution** If you are planning to run multiple shards on the same machine, then it is advisable to dedicate a separate drive to each shard. With many different threads trying to access different shards at the same time, it can over-stress a single drive, by forcing it to perform rapid read/write head movements between shards. By confining each shard on its own drive, you can minimize the movement of the heads and improve the overall performance considerably.

Summary

Sharding enables you to scale your datastores to handle extremely large datasets. It also enables you to grow the cluster to match the growth in your system.

MongoDB provides a simple automatic sharding configuration that works well for most requirements. Even though this process is automated, you can still fine-tune its characteristics to support your specific needs.

Auto-sharding is one of the key features of MongoDB that set it apart from other data-storage technologies. We hope this book has helped you see the many ways that MongoDB is designed to cope better with the rigorous demands of modern web-based applications than is possible using more traditional database tools.

Topics you have learned about in this book include the following:

- How to install and configure MongoDB on a variety of platforms.
- How to access MongoDB from various development languages.
- How to connect with the community surrounding the product, including how to obtain help and advice.
- How to design and build applications that take advantage of MongoDB's unique strengths.
- How to optimize, administer, and troubleshoot MongoDB-based datastores.
- How to create scalable fault-tolerant installations that span multiple servers.

You are strongly encouraged to explore the many samples and examples provided in this book. Other PHP examples can be found in the PHP MongoDB driver documentation located at `www.php.net/manual/en/book.mongo.php`. MongoDB is an extremely approachable tool, and its ease of installation and operation encourage experimentation. So don't hold back: crank it up and start playing with it! And remarkably soon you too will begin to appreciate all the possibilities that this intriguing product opens up for your applications.