13 Deployment and administration

This chapter covers

- Provisioning and hardware requirements
- Monitoring and diagnostics
- Backups and administrative tasks
- Security
- Performance troubleshooting
- Deployment checklist

This book would be incomplete without a few notes on deployment and administration. After all, it's one thing to use MongoDB but quite another to keep it running smoothly in production. The goal of this final chapter is to prepare you to make good decisions when deploying and administering MongoDB. You can think of this chapter as providing the wisdom required to keep you from experiencing the unpleasantness of a production database meltdown.

We'll start this chapter with MongoDB's hardware requirements, as well as some options for deploying this hardware. Then, we'll move into a few sections that discuss how to keep your system running, resilient, and secure. Finally, we'll end with a deployment checklist you can look back on to make sure you've covered all your bases.

13.1 Hardware and provisioning

The first question you need to ask before you deploy MongoDB is, "What I should deploy it on?" If you ran an entire production cluster on a single laptop as we did earlier in this book, you'd be in big trouble. In this section, we'll discuss how to choose the right topology for your requirements, how different hardware affects MongoDB, and what options are available for provisioning this hardware.

13.1.1 Cluster topology

This section gives you some basic recommendations on cluster topologies, but for a more complete analysis of different deployment topologies in replicated and sharded setups, you can consult chapters 11 and 12.

Figure 13.1 shows the minimum setup for the three different cluster types, as well as when you'd want to upgrade to a different type of cluster.

In total, there are three different types of clusters in MongoDB:

- *Single node*—As you can see at the top of figure 13.1, MongoDB can be run as a single server to support testing and staging environments. But for production deployments, a single server isn't recommended, even if journaling is enabled. Having only one machine complicates backup and recovery, and when there's a server failure, there's nothing to fail over to. That said, if you don't need reliability and have a small enough data set, this is always an option.
- *Replica set*—As shown in the middle of figure 13.1, the minimum recommended deployment topology for a replica set is three nodes, at least two of which should be data-storing nodes rather than arbiters. A replica set is necessary for automatic failover, easier backups, and not having a single point of failure. Refer to chapter 10 for more details on replica sets.
- *Sharded cluster*—As you can see at the bottom of figure 13.1, the minimum recommended deployment for a sharded cluster has two shards because deploying a sharded cluster with only one shard would add additional overhead without any of the benefits of sharding. Each shard should also be a replica set and there should be three config servers to ensure that there's no single point of failure. Note that there are also two `mongos` processes. Loss of all `mongos` processes doesn't lead to any data loss, but it does lead to downtime, so we have two here as part of the minimum production topology to ensure high availability. A sharded cluster is necessary when you want to scale up the capacity of your cluster by pooling together the capacity of a number of less powerful commodity servers. Refer to chapter 12 for more details on sharded clusters.

Now that you have a high-level understanding of the types of clusters that are available, let's go into more specific details about the deployment of each individual server.

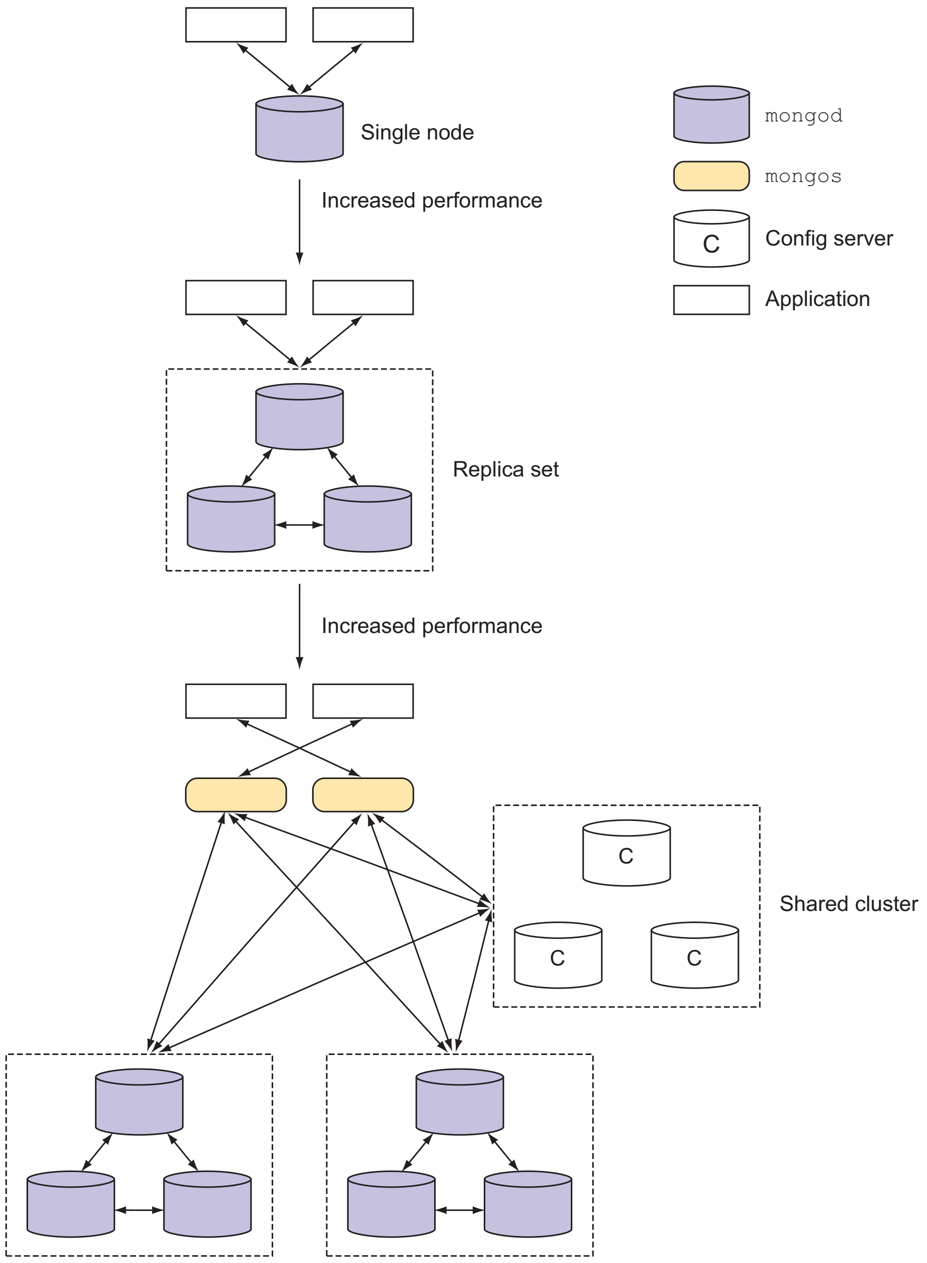

Figure 13.1 The minimum single node, replica set, and sharded cluster setups, as well as their purpose

13.1.2 *Deployment environment*

Here we'll present considerations for choosing good deployment environments for MongoDB. We'll discuss specific hardware requirements, such as CPU, RAM, and disks, and provide recommendations for optimizing the operating system environment. Figure 13.2 is a simplified visualization of the hardware and operating system components that MongoDB needs to interact with that we'll cover in this section. In the

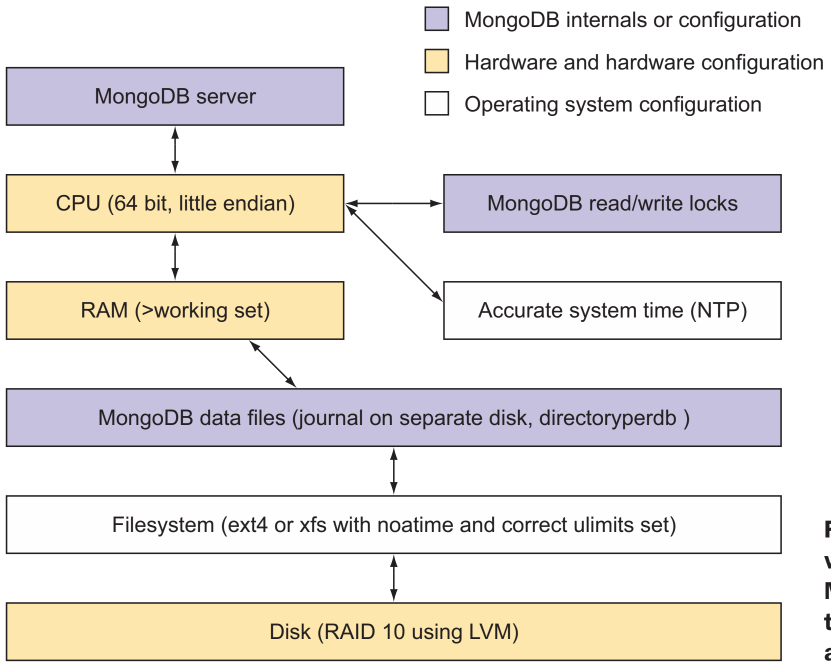
MongoDB internals or configuration
Hardware and hardware configuration
Operating system configuration
MongoDB server
CPU (64 bit, little endian)
MongoDB read/write locks
RAM (>working set)
Accurate system time (NTP)
MongoDB data files (journal on separate disk, directoryperdb)
Filesystem (ext4 or xfs with noatime and correct ulimits set)
Disk (RAID 10 using LVM)

Figure 13.2 A simplified visualization of how MongoDB depends on the operating system and hardware

subsequent sections we'll discuss cluster topology and provide some advice for deploying in the cloud.

ARCHITECTURE

Two notes on hardware architecture are in order. First, because MongoDB maps all data files to a virtual address space, all production deployments should be run on 64-bit machines. A 32-bit architecture limits MongoDB to about 2 GB of storage. With journaling enabled, the limit is reduced at around 1.5 GB. This is dangerous in production because if these limits are ever surpassed, MongoDB will behave unpredictably.[1] Feel free to run on 32-bit machines for unit testing and staging, but in production and for load testing, stick to 64-bit architectures.

Next, the components that ship with the MongoDB servers must be run on little-endian machines.[2] This usually isn't difficult to comply with since the x86 architecture, which you're likely using if you're not sure, is little endian, but users running SPARC, PowerPC, PA-RISC, and other big-endian architectures will have to hold off.[3] The client drivers, though, are maintained as separate projects, and they're usually built to support both little- and big-endian byte orderings. This means that even

[1] In this case "unpredictably" means that the behavior is completely undefined. This essentially means that MongoDB "might" simply crash, or it might make demons fly out of your nose. We just don't know. The main point here is that you should avoid this scenario because even the maintainers of MongoDB can't help you.

[2] The "endianness" of a machine is a hardware detail referring to the order in which bytes are stored in memory. See https://en.wikipedia.org/wiki/Endianness for more information.

[3] If you're interested in big-endian support for the core server, see https://jira.mongodb.org/browse/SERVER-1625.

though the server must run on little endian machines, clients of MongoDB can usually run on either architecture.

CPU

MongoDB isn't particularly CPU-intensive; database operations are rarely CPU-bound, so this isn't the first place to look when diagnosing a performance issue. Your first priority when optimizing for MongoDB is to ensure operations aren't I/O-bound (we'll discuss I/O-bound issues more in the next two sections on RAM and disks).

But once your indexes and working set fit entirely in RAM, you may see some CPU-boundedness. If you have a single MongoDB instance serving tens (or hundreds) of thousands of queries per second, you can realize performance increases by providing more CPU cores.

If you do happen to see CPU saturation, check your logs for slow query warnings. There may be some types of queries that are inherently more CPU-intensive, or you may have an index issue that the logs will help you diagnose. But if that's not the case and you're still seeing CPU saturation, it's likely due to lock contention, which we'll briefly touch on here.

RAM

As with any database, MongoDB performs best with sufficient RAM. Be sure to select hardware with enough RAM to contain your frequently used indexes, plus your working data set. Then as your data grows, keep a close eye on the ratio of RAM-to–working set size. If you allow working set size to grow beyond RAM, you may start to see significant performance degradation. Paging from disk in and of itself isn't a problem—it's a necessary step in loading data into memory. But if you're unhappy with performance, excessive paging may be your problem. (Chapter 8 discusses the relationship between working set, index size, and RAM in great detail.) By the end of this chapter, you should have enough tools in your arsenal to diagnose RAM deficiencies.

There are a few use cases where you can safely let data size grow well beyond available RAM, but they're the exception, not the rule. One example is using MongoDB as an archive, where reads and writes seldom happen and you don't need fast responses. Because RAM is essentially a cache for pages on disk, it only provides a performance boost if a page will be used more than once in a short period of time. In the case of an archive application, this may not be the case, and the page will need to be loaded into memory from disk the first (and possibly only) time it's used anyway. On top of that, having as much RAM as data for this type of application might be prohibitively expensive. For all data sets, the key is testing. Test a representative prototype of your application to ensure that you get the necessary baseline performance.

DISKS

When choosing disks, you need to consider cost, IOPS (input/output operations per second), seek time, and storage capacity. The differences between running on a single consumer-grade hard drive, running in the cloud in a virtual disk (say, EBS), and running against a high-performance SAN can't be overemphasized. Some applications will perform acceptably against a single network-attached EBS volume, but demanding applications will require something more.

Disk performance is important for a few reasons:

- *High write workloads*—As you're writing to MongoDB, the server must flush the data back to disk. With a write-intensive app and a slow disk, the flushing may be slow enough to negatively affect overall system performance.
- *A fast disk allows a quicker server warm-up*—Any time you need to restart a server, you also have to load your data set into RAM. This happens lazily; each successive read or write to MongoDB will load a new virtual memory page into RAM until the physical memory is full. A fast disk will make this process much faster, which will increase MongoDB's performance following a cold restart.
- *A fast disk alters the required ratio of working set size to RAM for your application*—Using, say, a solid-state drive, you may be able to run with much less RAM (or much greater capacity) than you would otherwise.

Regardless of the type of disk used, serious deployments will generally use, not a single disk, but a redundant array of disks (RAID) instead. Users typically manage a RAID cluster using Linux's logical volume manager, LVM, with a RAID level of 10. RAID 10 provides redundancy while maintaining acceptable performance, and it's commonly used in MongoDB deployments.[4] Note that this is more expensive than a single disk, which illustrates the tradeoff between cost and performance. Even more advanced deployments will use a high-performance self-managed SAN, where the disks are all virtual and the idea of RAID may not even apply.

If your data is spread across multiple databases within the same MongoDB server, then you can also ensure capacity by using the server's `--directoryperdb` flag. This will create a separate directory in the data file path for each database. Using this, you can conceivably mount a separate volume (RAID'ed or not) for each database. This may allow you to take advantage of some performance increases, because you'll be able to read from separate sets of spindles (or solid-state drives).

Locks

MongoDB's locking model is a topic unto itself. We won't discuss all the nuances of concurrency and performance in MongoDB here, but we'll cover the basic concurrency models MongoDB supports. In practice, ensuring that you don't have lock contention will require careful monitoring or benchmarking because every workload is different and may have completely different points of contention.

In the early days, MongoDB took a global lock on the entire server. This was soon updated to a global lock on each database and support was added to release the lock before disk operations.

As of v3.0 MongoDB now supports two separate storage engines with different concurrency models. It has collection-level locking in its own native mmap-based storage

[4] For an overview of RAID levels, see http://en.wikipedia.org/wiki/Standard_RAID_levels.

engine, and document-level locking in the newly supported WiredTiger storage engine, which can be used instead of the native storage engine. Consult JIRA, MongoDB's bug-tracking system, as well as the latest release notes for the status of these improvements.[5]

FILESYSTEMS

You'll get the best performance from MongoDB if you run it on the right filesystem. Two in particular, ext4 and xfs, feature fast, contiguous disk allocation. Using these filesystems will speed up MongoDB's frequent preallocations.[6]

Once you mount your fast filesystem, you can achieve another performance gain by disabling updates to files' last access time: `atime`. Normally, the operating system will update a file's `atime` every time the file is read or written. In a database environment, this amounts to a lot of unnecessary work. Disabling `atime` on Linux is relatively easy:

1. First, make a backup of the filesystem config file:

   ```
   sudo cp /etc/fstab /etc/fstab.bak
   ```

2. Open the original file in your favorite editor:

   ```
   sudo vim /etc/fstab
   ```

3. For each mounted volume you'll find inside /etc/fstab, you'll see a list of settings aligned by column. Under the options column, add the `noatime` directive:

   ```
   # file-system       mount  type    options  dump  pass
   UUID=8309beda-bf62-43  /ssd   ext4    noatime  0     2
   ```

4. Save your work. The new settings should take effect immediately.[7]

You can see the list of all mounted filesystems with the help of the `findmnt` command, which exists on Linux machines:

```
$ findmnt -s
TARGET SOURCE    FSTYPE OPTIONS
/proc  proc      proc   defaults
/      /dev/xvda ext3   noatime,errors=remount-ro
none   /dev/xvdb swap   sw
```

The `-s` option makes `findmnt` get its data from the /etc/fstab file. Running `findmnt` without any command-line parameters shows more details yet more busy output.

FILE DESCRIPTORS

Some Linux systems cap the number of allowable open file descriptors at 1024. This is occasionally too low for MongoDB and may result in warning messages or even errors

[5] Release notes can be found at https://jira.mongodb.org/browse/server and https://docs.mongodb.org/manual/release-notes/ for more information.

[6] For more detailed filesystem recommendations, see https://docs.mongodb.org/manual/administration/production-notes/#kernel-and-file-systems.

[7] Note that these are basic recommendations and by no means comprehensive. See https://en.wikipedia.org/wiki/Fstab for an overview of mount options.

when opening connections (which you'll see clearly in the logs). Naturally, MongoDB requires a file descriptor for each open file and network connection.

Assuming you store your data files in a folder with the word "*data*" in it, you can see the number of data file descriptors using `lsof` and a few well-placed pipes:

```
lsof | grep mongo | grep data | wc -l
```

Counting the number of network connection descriptors is just as easy:

```
lsof | grep mongo | grep TCP | wc -l
```

When it comes to file descriptors, the best strategy is to start with a high limit so that you never run out in production. You can check the current limit temporarily with the `ulimit` command:

```
ulimit -Hn
```

To raise the limit permanently, open your limits.conf file with your editor of choice:

```
sudo vim /etc/security/limits.conf
```

Then set the soft and hard limits. These are specified on a per-user basis. This example assumes that the `mongodb` user will run the `mongod` process:

```
mongodb soft nofile 2048
mongodb hard nofile 10240
```

The new settings will take effect when that user logs in again.[8]

Clocks

It turns out that replication is susceptible to "clock skew," which can occur if the clocks on the machines hosting the various nodes of a replica set get out of sync. Replication depends heavily on time comparisons, so if different machines in the same replica set disagree on the current time, that can cause serious problems. This isn't ideal, but fortunately there's a solution. You need to ensure that each of your servers uses NTP (Network Time Protocol)[9] or some other synchronization protocol to keep their clocks synchronized:

- On Unix variants, this means running the `ntpd` daemon.
- On Windows, the Windows Time Services fulfills this role.

Journaling

MongoDB v1.8 introduced journaling, and since v2.0 MongoDB enables journaling by default. When journaling is enabled, MongoDB will commit all writes to a journal

[8] This may not apply if the user is purely a daemon. In that case restarting the `mongodb` service using something like `sudo service mongodb restart` should do the trick.

[9] Make sure to verify that NTP is working correctly using diagnostic commands such as `ntpstat`.

before writing to the core data files. This allows the MongoDB server to come back online quickly and cleanly in the event of an unclean shutdown.

In the event of an unclean shutdown of a nonjournaled `mongod` process, restoring the data files to a consistent state requires running a repair. The repair process rewrites the data files, discarding anything it can't understand (corrupted data). Because downtime and data loss are generally frowned upon, repairing in this way is usually part of a last-ditch recovery effort. Resyncing from an existing replica is almost always easier and more reliable. Being able to recover in this way is one of the reasons why it's so important to run with replication.

Journaling obviates the need for database repairs because MongoDB can use the journal to restore the data files to a consistent state. In MongoDB v2.0 as well as v3.0, journaling is enabled by default, but you can disable it with the `--nojournal` flag:

```
$ mongod --nojournal
```

When enabled, the journal files will be kept in a directory called journal, located just below the main data path.

If you run your MongoDB server with journaling enabled, keep a of couple points in mind:

- First, journaling adds some additional overhead to writes.
- One way to mitigate this is to allocate a separate disk for the journal, and then either create a symlink[10] between the journal directory and the auxiliary volume or simply mount this disk where the journal directory should be. The auxiliary volume needn't be large; a 120 GB disk is more than sufficient, and a solid-state drive (SSD) of this size is affordable. Mounting a separate SSD for the journal files will ensure that journaling runs with the smallest possible performance penalty.
- Second, journaling by itself doesn't guarantee that no write will be lost. It guarantees only that MongoDB will always come back online in a consistent state. Journaling works by syncing a write buffer to disk every 100 ms, so an unclean shutdown can result in the loss of up to the last 100 ms of writes. If this isn't acceptable for any part of your application, you can change the write concern of operations through any client driver. You'd run this as a safe mode option (just like `w` and `wtimeout`). For example, in the Ruby driver, you might use the `j` option like this in order to have safe mode enabled all the time for one of the servers:

  ```
  client = Mongo::Client.new( ['127.0.0.1:27017'], :write => {:j =>
  true}, :database => 'garden')
  ```

[10] A symlink, or symbolic link, is essentially an object that looks like a file but is actually a reference to another location. Using this here means that MongoDB can still find the journal directory in the same place despite the fact that it's on a completely different disk. See https://en.wikipedia.org/wiki/Symbolic_link for more details.

Be aware that running this after every write is unwise because it forces every write to wait for the next journal sync.[11] Even then, these writes are subject to rollbacks in a replicated setup because journal acknowledged writes only wait until the primary writes to its journal and doesn't wait for replication to secondaries. Therefore, a more commonly used way to ensure your writes are durable is to use the "majority" write concern option. This ensures that the write reaches a majority of the replica set, which is usually sufficient and resilient to failures. See chapter 10 for more details on replication and write concern options.[12]

13.1.3 Provisioning

You can certainly deploy on your own hardware, and that approach does have its advantages, but this section will focus mostly on cloud deployments and automation. In-house deployments require a lot of specialized administration and setup that's outside the scope of this book.

The cloud

More and more users are running MongoDB in hosted, virtualized environments, collectively known as the cloud. Among these, Amazon's EC2 has become a deployment environment of choice because of its ease of use, wide geographic availability, and competitive pricing. EC2, and other environments like it, can be adequate for deploying MongoDB. At a high level, there are three main components to consider when thinking about deploying on any cloud provider:

- The hosts themselves, which will actually run MongoDB
- The persistent storage system, which will store MongoDB's data files
- The network that MongoDB will use to communicate internally and with clients

First, EC2 hosts are convenient, because they're easy to provision and can be allocated as you need them, so the cost will scale with your needs. But one disadvantage of an EC2 host is that it's essentially a black box. You may experience service blips or instance slowdowns and have no way of diagnosing or remedying them. Sometimes you might get a "bad box," which means your virtual instance got allocated on a machine with slow hardware or several other active users.[13]

Second, EC2 allows you to mount virtual block devices known as EBS volumes as your persistent storage. EBS volumes provide a great deal of flexibility, allowing you to add storage and move volumes across machines as needed. EBS also lets you take snapshots, which you can use for backups.

---

[11] Recent versions of MongoDB allow changing the journal commit interval to reduce the latency on journal-acknowledged writes, using the `commitIntervalMs` option.

[12] For more details on the behavior of specific write concern levels, see https://docs.mongodb.org/manual/core/write-concern.

[13] Note that EC2 has many different instance types and purchasing options that may suffer more or less from these problems. Check with EC2 or your cloud provider to find out exactly how they provision your machine.

The problem with the cheapest EBS volumes is that they may not provide a high level of throughput compared to what's possible with physical disks. Be sure to check with your hosting provider to learn what their current offerings are. As of this writing, EBS volumes can be SSD-backed, and Amazon also provides an option to use disks local to the node. On top of that, another way to increase performance is to run whatever disks you're using with a RAID 10 for increased read throughput.

Finally, EC2's network is sufficient for most users, but like the storage and the hosts themselves, you give up a great deal of control by using EC2's network rather than one you set up yourself. You're sharing the same network with many others, so high traffic from other applications have the potential to affect your network performance. Like disks and instance types, this is "pay to play" when dealing with a hosting provider, so you may be able to pay more to mitigate these problems.

In summary, deploying on EC2 has many advantages, but it clearly has some notable drawbacks. For these reasons, rather than dealing with some of EC2's limitations and unpredictability, many users prefer to run MongoDB on their own physical hardware. Then again, EC2 and the cloud in general are convenient and perfectly acceptable for a wide variety of users. In the end, the best way to approach deploying in any environment is careful testing. Benchmarking the cloud platform that you want to deploy on will ultimately help you make the decision more than anything else.

MMS AUTOMATION

Another provisioning option that is relatively new is the MongoDB Management System (MMS) Automation. MMS Automation can provision instances from EC2 and setup of your entire cluster with the press of a button.[14] The MMS team at MongoDB is constantly adding new features to simplify MongoDB operations, so if you're interested in using this, refer to the current documentation at https://docs.mms.mongodb.com to find out what's available.

13.2 Monitoring and diagnostics

Once you've deployed MongoDB in production, you'll want to keep an eye on it. If performance is slowly degrading, or if failures are occurring frequently, you'll want to be apprised of these. That's where monitoring comes in.

Let's start with the simplest kind of monitoring: logging. Then we'll explore the built-in commands that provide the most information about the running MongoDB server; these commands underlie the `mongostat` utility and the web console, both of which we'll describe in brief.

Then we'll look at the MMS Monitoring provided by the MongoDB company. We'll make a couple of recommendations on external monitoring tools and end the section by presenting two diagnostic utilities: `bsondump` and `mongosniff`.

[14] MMS Automation can also deploy MongoDB onto preprovisioned hardware if you're not using EC2.

13.2.1 Logging

Logging is the first level of monitoring; as such, you should plan on keeping logs for all your deployments. This usually isn't a problem because MongoDB requires that you specify the `--logpath` option when running it in the background. But there are a few extra settings to be aware of. To enable verbose logging, start the `mongod` process with the `-vvvvv` option (the more vs, the more verbose the output). This is handy if, for instance, you need to debug some code and want to log every query. But do be aware that verbose logging will make your logs quite large and may affect server performance. If your logs become too unwieldy, remember that you can always store your logs on a different partition.

Next you can start `mongod` with the `--logappend` option. This will append to an existing log rather than moving it and appending a timestamp to the filename, which is the default behavior.

Finally, if you have a long-running MongoDB process, you may want to write a script that periodically rotates the log files. MongoDB provides the `logrotate` command for this purpose. Here's how to run it from the shell:

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

Sending the SIGUSR1[15] signal to the process also runs the `logrotate` command. Here's how to send that signal to process number 12345:

```
$ kill -SIGUSR1 12345
```

You can find the process ID of the process you want to send the signal to using the `ps` command, like this:

```
$ ps –ef | grep mongo
```

Note that the `kill` command isn't always as dire as it sounds. It only sends a signal to a running process, but was named in the days when most or all signals ended the process.[16] But running `kill` with the `-9` command-line option will end a process in a brutal way and should be avoided as much as possible on production systems.

13.2.2 MongoDB diagnostic commands

MongoDB has a number of database commands used to report internal state. These underlie all MongoDB monitoring applications. Here's a quick reference for a few of the commands that you might find useful:

- Global server statistics: `db.serverStatus()`
- Stats for currently running operation: `db.currentOp()`

[15] Unix-like systems support sending "signals" to running processes, which can trigger certain actions if the process that receives the signal is written to handle it. MongoDB is configured to handle receiving the SIGUSR1 signal and rotates the logs when it receives it.

[16] There's no Windows equivalent for this, so you'll have to rely on the `logrotate` command in that case.

- Include stats for idle system operations: `db.currentOp(true)`
- Per database counters and activity stats: `db.runCommand({top:1})`
- Memory and disk usage statistics: `db.stats()`

The output for all of these commands improves with each MongoDB release, so documenting it in a semi-permanent medium like this book isn't always helpful. Consult the documentation for your version of MongoDB to find out what each field in the output means.

13.2.3 MongoDB diagnostic tools

In addition to the diagnostic commands listed previously, MongoDB ships with a few handy diagnostic tools. Most of these are built on the previous commands and could be easily implemented using a driver or the mongo shell.

Here's a quick introduction to what we'll cover in this section:

- `mongostat`—Global system statistics
- `mongotop`—Global operation statistics
- `mongosniff` (advanced)—Dump MongoDB network traffic
- `bsondump`—Display BSON files as JSON

MONGOSTAT

The `db.currentOp()` method shows only the operations queued or in progress at a particular moment in time. Similarly, the `serverStatus` command provides a point-in-time snapshot of various system fields and counters. But sometimes you need a view of the system's real-time activity, and that's where `mongostat` comes in. Modeled after `iostat` and other similar tools, `mongostat` polls the server at a fixed interval and displays an array of statistics, from the number of inserts per second to the amount of resident memory, to the frequency of B-tree page misses.

You can invoke the `mongostat` command on localhost, and the polling will occur once a second:

```
$ mongostat
```

It's also highly configurable, and you can start it with `--help` to see all the options. For example, you can use the `--host` option to connect to a host and port besides the default of `localhost:27017`. One of the more notable features is cluster discovery; when you start `mongostat` with the `--discover` option, you can use the `--host` option to point it to a single node, and it'll discover the remaining nodes in a replica set or sharded cluster. It then displays the entire cluster's statistics in aggregate.

MONGOTOP

Similar to the way `mongostat` is the external tool for the `db.currentOp()` and `serverStatus` commands, `mongotop` is the external tool for the `top` command. You can run this in exactly the same way as `mongostat`, assuming you have a server running on the local machine and listening on the default port:

```
$ mongotop
```

As with mongostat, you can run this command with -help to see a number of useful configuration options.

MONGOSNIFF

The next command we'll cover is mongosniff, which sniffs packets from a client to the MongoDB server and prints them intelligibly. If you happen to be writing a driver or debugging an errant connection, then this is your tool. You can start it up like this to listen on the local network interface at the default port:

```
sudo mongosniff --source NET I0
```

Then when you connect with any client—say, the MongoDB shell—you'll get an easy-to-read stream of network chatter:

```
127.0.0.1:58022  -->> 127.0.0.1:27017 test.$cmd 61 bytes id:89ac9c1d
2309790749 query: { isMaster: 1.0 }  ntoreturn: -1
127.0.0.1:27017  <<--  127.0.0.1:58022   87 bytes
reply n:1 cursorId: 0 { ismaster: true, ok: 1.0 }
```

Here you can see a client running the isMaster command, which is represented as a query for { isMaster: 1.0 } against the special test.$cmd collection. You can also see that the response document contains ismaster: true, indicating that the node that this command was sent to was in fact the primary. You can see all the mongosniff options by running it with --help.

BSONDUMP

Another useful command is bsondump, which allows you to examine raw BSON files. BSON files are generated by the mongodump command (discussed in section 13.3) and by replica set rollbacks.[17] For instance, let's say you've dumped a collection with a single document. If that collection ends up in a file called users.bson, then can examine the contents easily:

```
$ bsondump users.bson
{ "_id" : ObjectId( "4d82836dc3efdb9915012b91" ), "name" : "Kyle" }
```

As you can see, bsondump prints the BSON as JSON by default. If you're doing serious debugging, you'll want to see the real composition of BSON types and sizes. For that, run the command in debug mode:

```
$ bsondump --type=debug users.bson
--- new object ---
size : 37
_id
type:  7 size: 17
name
type:  2 size: 15
```

[17] MongoDB writes replica set rollback files to the rollback directory inside the data directory.

This gives you the total size of the object (37 bytes), the types of the two fields (7 and 2), and those fields' sizes.

THE WEB CONSOLE

Finally, MongoDB provides some access to statistics via a web interface and a REST server. As of v3.0, these systems are old and under active development. On top of that, they report the same information available via the other tools or database commands presented earlier. If you want to use these systems, be sure to look at the current documentation and carefully consider the security implications.

13.2.4 MongoDB Monitoring Service

MongoDB, Inc. provides MMS Monitoring for free, which not only allows you to view dashboards to help you understand your system, but also provides an easy way to share your system information with MongoDB support, which is indispensable if you ever need help with your system. MMS Monitoring can also be licensed as a self-hosted version for large enterprises with paid contracts. To get started, all you need to do is create an account on the MMS Monitoring website at https://mms.mongodb.com. Once you create an account, you'll see instructions to walk you through the process of setting up MMS, which we won't cover here.

13.2.5 External monitoring applications

Most serious deployments will require an external monitoring application. Nagios and Munin are two popular open source monitoring systems used to keep an eye on many MongoDB deployments. You can use each of these with MongoDB by installing a simple open source plug-in.

Writing a plug-in for any arbitrary monitoring application isn't difficult. It mostly involves running various statistics commands against a live MongoDB database. The `serverStatus`, `dbstats`, and `collstats` commands usually provide all the information you might need, and you can get all of them straight from the HTTP REST interface, avoiding the need for a driver.

Finally, don't forget the wealth of tools available for low-level system monitoring. For example, the `iostat` command can be helpful in diagnosing MongoDB performance issues. Most of the performance issues in MongoDB deployments can be traced to a single source: the hard disk.

In the following example, we use the `-x` option to show extended statistics and specify `2` to display those stats at two-second intervals:

```
$ iostat -x 2
Device:  rsec/s   wsec/s avgrq-sz avgqu-sz   await  svctm  %util
sdb        0.00  3101.12    10.09    32.83  101.39   1.34  29.36
Device:  rsec/s   wsec/s avgrq-sz avgqu-sz   await  svctm  %util
sdb        0.00  2933.93     9.87    23.72  125.23   1.47  34.13
```

For a detailed description of each of these fields, or for details on your specific version of `iostat`, consult your system's man[18] pages. For a quick diagnostic, you'll be most interested in two of the columns shown:

- The `await` column indicates the average time in milliseconds for serving I/O requests. This average includes time spent in the I/O queue and time spent actually servicing I/O requests.
- `%util` is the percentage of CPU during which I/O requests were issued to the device, which essentially translates to the bandwidth use of the device.

The preceding `iostat` snippet shows moderate disk usage. The average time waiting on I/O is around 100 ms (hint: that's a lot!), and the utilization is about 30%. If you were to investigate the MongoDB logs on this machine, you'd likely see numerous slow operations (queries, inserts, or otherwise). In fact, it's those slow operations that would initially alert you to a potential problem. The `iostat` output can help you confirm the problem. We've covered a lot of ways to diagnose and monitor a running system to keep it running smoothly, but now let's get into an unavoidable aspect of a production deployment: backups.

13.3 Backups

Part of running a production database deployment is being prepared for disasters. Backups play an important role in this. When disaster strikes, a good backup can save the day, and in these cases, you'll never regret having invested time and diligence in a regular backup policy. Yet some users still decide that they can live without backups. These users have only themselves to blame when they can't recover their databases. Don't be one of these users.

Three general strategies for backing up a MongoDB database are as follows:

- Using `mongodump` and `mongorestore`
- Copying the raw data files
- Using MMS Backups

We'll go over each of these strategies in the next three sections.

13.3.1 mongodump and mongorestore

`mongodump` writes the contents of a database as BSON files. `mongorestore` reads these files and restores them. These tools are useful for backing up individual collections and databases as well as the whole server. They can be run against a live server (you don't have to lock or shut down the server), or you can point them to a set of data files,

[18] The man pages, or "manual" pages, are the way many programs on Unix-based systems provide documentation. For example, type man iostat in a terminal to get the iostat man page.

but only when the server is locked or shut down. The simplest way to run mongodump is like this:[19]

```
$ mongodump -h localhost --port 27017
```

This will dump each database and collection from the server at localhost to a directory called dump.[20] The dump directory will include all the documents from each collection, including the system collections that define users and indexes. But significantly, the indexes themselves won't be included in the dump. This means that when you restore, any indexes will have to be rebuilt. If you have an especially large data set, or a large number of indexes, this will take time.

RESTORING BSON FILES

To restore BSON files, run mongorestore and point it at the dump folder:

```
$ mongorestore -h localhost --port 27017 dump
```

Note that when restoring, mongorestore won't drop data by default, so if you're restoring to an existing database, be sure to run with the --drop flag.

13.3.2 Data file–based backups

Most users opt for a file-based backup, where the raw data files are copied to a new location. This approach is often faster than mongodump because the backups and restorations require no transformation of the data.

The only potential problem with a file-based backup is that it requires locking the database, but generally you'll lock a secondary node and thus should be able to keep your application online for the duration of the backup.

COPYING THE DATA FILES

Users frequently make the mistake of copying the data files without first locking the database. Even if journaling is enabled, doing so will result in corruption of the copied files. This section will cover how to bring the data files to a consistent state by locking the database to allow for safe backups.

Snapshotting a live system

You may wonder why you have to lock the data files even when journaling is enabled. The answer is that journaling is only able to restore a database to a consistent state from a single point in time. If you're manually copying the data files, there may be some delay between copying each file, which means different data files may be from different points in time, and journaling can't deal with this.

[19] If you have secondary reads enabled, you can run this against the secondary of a replica set rather than the primary.

[20] If possible, this directory should be stored on a separate disk, both for performance and for extra defense against disk failure.

But if your filesystem, storage provider, or hosting provider explicitly supports "point-in-time" snapshots, you can use this feature to safely snapshot a live system without locking the data files. Note that everything has to be saved at exactly the same time, including the journal. This means that if your journal is on a separate disk (or you have journaling disabled), you're out of luck unless you have a system that supports point-in-time snapshots across multiple volumes.

To safely copy the data files, you first need to make sure that they're in a consistent state, so you either have to shut down the database or lock it. Because shutting down the database might be too involved for some deployments, most users opt for the locking approach. Here's the command for syncing and locking:

```
> use admin
> db.fsyncLock()
```

At this point, the database is locked against writes[21] and the data files are synced to disk. This means that it's now safe to copy the data files. If you're running on a filesystem or storage system that supports snapshots, it's best to take a snapshot and copy later. This allows you to unlock quickly.

If you can't run a snapshot, you'll have to keep the database locked while you copy the data files. If you're copying data files from a secondary node, be sure that the node is current with the primary and has enough oplog to remain offline for the duration of the backup.

Once you've finished making a snapshot or backing up, you can unlock the database. The somewhat arcane unlock command can be issued like this:

```
> db.fsyncUnlock()
```

Note that this is merely a request to unlock; the database may not unlock right away. Run the `db.currentOp()` method to verify that the database is no longer locked.

13.3.3 MMS backups

Once again, the MMS team at MongoDB has a solution for this problem. MMS Backups use the replication oplog to provide point-in-time backups for your entire cluster. This works for stand-alone replica sets as well as for entire sharded cluster. As we mentioned earlier, the MMS team is constantly adding new features, so check the up-to-date documentation for details.

[21] Any attempts to write will block behind this lock, and reads that come after those writes will also block. This unfortunately includes reads of authentication data, meaning that new connection attempts will also block on this lock when you're running with authentication, so keep your connections open when you're running the backup process. See https://docs.mongodb.org/manual/reference/method/db.fsyncLock/ for more details.

13.4 Security

Security is an extremely important, and often overlooked, aspect of deploying a production database. In this section, we'll cover the main types of security, including secure environments, network encryption, authentication, and authorization.

We'll end with a brief discussion of which security features are only available in the enterprise edition of MongoDB. Perhaps more than for any other topic, it's vital to stay up to date with the current security tools and best practices, so treat this section as an overview of what to consider when thinking about security, but consult the most recent documentation at https://docs.mongodb.org/manual/security when putting it into production.

13.4.1 Secure environments

MongoDB, like all databases, should be run in a secure environment. Production users of MongoDB must take advantage of the security features of modern operating systems to ensure the safety of their data. Probably the most important of these features is the firewall.

The only potential difficulty in using a firewall with MongoDB is knowing which machines need to communicate with each other. Fortunately, the communication rules are simple:

- With a replica set, each node must be able to reach every other node.
- All database clients must be able to connect with every replica set node that the client might conceivably talk to.
- All communication is done using the TCP protocol.
- For a node to be reachable, it means that it's reachable on the port that it was configured to listen on. For example, `mongod` listens on TCP port 27017 by default, so to be reachable it must be reachable on that port.

A shard cluster consists in part of replica sets. All the replica set rules apply; the client in the case of sharding is the `mongos` router. Additionally:

- All shards must be able to communicate directly with one another.
- Both the shards and the `mongos` routers must be able to talk to the config servers.

Figure 13.2 shows a simplified visualization of these connectivity rules, with the one addition that any arrow connecting to a boxed replica set or set of config servers means that the connectivity requirement applies to every individual server inside the box.

For the most part, running MongoDB in a secure environment is completely external to MongoDB and is a full topic on its own. But one option, `--bind_ip`, is relevant here.[22] By default, MongoDB will listen on all addresses on the machine, B=but you may want MongoDB to listen on one or more specific addresses instead. For this you

[22] Note that as of v2.6, the prebuilt packages for Linux include `--bind_ip` by default.

can start `mongod` and `mongos` with the `--bind_ip` option, which takes a list of one or more comma-separated IP addresses. For example, to listen on the loopback interface as well as on the internal IP address 10.4.1.55, you'd start `mongod` like this:

```
mongod --bind_ip 127.0.0.1,10.4.1.55
```

Note that data between machines will be sent in the clear unless you have SSL enabled, which we'll cover in the next section.

13.4.2 Network encryption

Perhaps the most fundamental aspect of securing your system is ensuring your network traffic is encrypted. Unless your system is completely isolated and no one you don't trust can even see your traffic (for example, if all your traffic is already encrypted over a virtual private network, or your network routing rules are set up such that no traffic can be sent to your machines from outside your trusted network[23]), you should probably use MongoDB with encryption. Fortunately, as of v2.4, MongoDB ships with a library that handles this encryption—called the Secure Sockets Layer (SSL)—built in.

To see why this is important, let's play the role of an eavesdropper using the Unix command `tcpdump`. First, use the `ifconfig` command to find the name of the loopback interface, or the interface that programs communicating from the local machine to the local machine use.

Here's what the beginning of the output looks like on our machine:

```
$ ifconfig
lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        ...
        ...
```

For us, the loopback interface is `lo`. Now we can use the appropriate `tcpdump` command to dump all traffic on this interface:

```
$ sudo tcpdump -i lo -X
```

NOTE Reading network traffic using `tcpdump` requires root permissions, so if you can't run this command, just read along with the example that follows.

In another terminal on the same machine, start a `mongod` server without SSL enabled (change the data path as appropriate):

```
$ mongod --dbpath /data/db/
```

[23] Many cloud hosting providers have tools to help with this, such as Virtual Private Clouds (VPCs), subnets, and security groups in AWS.

Then, connect to the database and insert a single document:

```
$ mongo
...
> db.test.insert({ "message" : "plaintext" }) > exit
bye
```

Now, if you look at the tcpdump output in the terminal, you'll see a number of packets output, one of which looks something like this:

```
16:05:10.507867 IP localhost.localdomain.50891 >
    ➥ localhost.localdomain.27017 …
    0x0000:  4500 007f aa4a 4000 4006 922c 7f00 0001  E....J@.@..,....
    0x0010:  7f00 0001 c6cb 6989 cf17 1d67 d7e6 c88f  . ....i....g....
    0x0020:  8018 0156 fe73 0000 0101 080a 0018 062e  ...V.s. ........
    0x0030:  0017 b6f6 4b00 0000 0300 0000 ffff ffff  ....K. .........
    0x0040:  d207 0000 0000 0000 7465 7374 2e74 6573  . ......test.tes
    0x0050:  7400 2d00 0000 075f 6964 0054 7f7b 0649  t.-...._id.T.{.I
    0x0060:  45fa 2cfc 65c5 8402 6d65 7373 6167 6500  E.,.e...message.
    0x0070:  0a00 0000 706c 6169 6e74 6578 7400 00    ....plaintext..
```

Document sent in plaintext over the network (1)

There's our message, right in the clear (1)! This shows how important network encryption is. Now, let's run MongoDB with SSL and see what happens.

RUN MONGODB WITH SSL

First, generate the key for the server:

```
openssl req -newkey rsa:2048 -new -x509 -days 365 -nodes -out mongodbcert.crt
-keyout mongodb-cert.key
cat mongodb-cert.key mongodb-cert.crt > mongodb.pem
```

Then, run the mongod server with SSL, using the --sslPEMKeyFile and --sslMode options:

```
$ mongod --sslMode requireSSL --sslPEMKeyFile mongodb.pem
```

Now, connect the client with SSL and do exactly the same operation:

```
$ mongo --ssl
...
> db.test.insert({ "message" : "plaintext" }) > exit
bye
```

If you now go back to the window with tcpdump, you'll see something completely incomprehensible where the message used to be:

```
16:09:26.269944 IP localhost.localdomain.50899 >
  ➥ localhost.localdomain.27017: …
    0x0000:  4500 009c 52c3 4000 4006 e996 7f00 0001  E...R.@.@. .....
    0x0010:  7f00 0001 c6d3 6989 c46a 4267 7ac5 5202  . ....i..jBgz.R.
    0x0020:  8018 0173 fe90 0000 0101 080a 001b ed40  ...s. .........@
```

Document sent over the network securely encrypted (1)

```
0x0030:  001b 6c4c 1703 0300 637d b671 2e7b 499d  ..lL....c}.q.{I.
0x0040:  3fe8 b303 2933 d04b ff5c 3ccf fac2 023d  ?...)3.K.\<....=
0x0050:  b2a1 28a0 6d3f f215 54ea 4396 7f55 f8de  ..(.m?..T.C..U..
0x0060:  bb8d 2e20 0889 f3db 2229 1645 ceed 2d20  . ......").E..-.
0x0070:  1593 e508 6b33 9ae1 edb5 f099 9801 55ae  ....k3. ......U.
0x0080:  d443 6a65 2345 019f 3121 c570 3d9d 31b4  .Cje#E..1!.p=.1.
0x0090:  bf80 ea12 e7ca 8c4e 777a 45dd            . .....NwzE.
```

Success! Is our system secure now? Not quite. Proper encryption is only one aspect of securing your system. In the next section, we'll discuss how to verify the identity of services and users, and following that we'll talk about how to get fine-grained control of what each user is allowed to do.

SSL in clusters

Now you know how to set up SSL between the mongo shell and a single `mongod`, which raises the obvious question of how this extends to an entire cluster. Fortunately this is fairly similar to the previous example. Simply start every node in the cluster with the `--sslMode requireSSL` option. If you're already running a cluster without SSL, there's an upgrade process to ensure that you can upgrade without losing connectivity between your nodes in the transition; see https://docs.mongodb.org/manual/tutorial/upgrade-cluster-to-ssl/.

Keep your keys safe!

Most of the security mechanisms in this chapter depend on the exchange of keys. Make sure to keep your keys safe (and stored using the proper permissions) and not share them between servers except when necessary.

As an extreme example, if you use the same key across all machines, that means all an attacker has to do is compromise that single key to read all traffic on your network.

If you're ever unsure about whether you should share a key, consult the official documentation either for MongoDB or the underlying mechanism.

13.4.3 Authentication

The next layer of security is authentication. What good is network encryption if anyone on the internet can pose as a legitimate user and do whatever they want with your system?

Authentication allows you to verify the identity of services and users in a secure way. First we'll discuss why and how to authenticate services and then users. We'll then briefly discuss how these concepts translate to replica sets and sharded clusters. As always, we'll cover the core concepts here, but you should consult the latest documentation for your version of MongoDB to ensure you're up-to-date (https://docs.mongodb.org/manual/core/authentication).

SERVICE AUTHENTICATION

The first stage of authentication is verifying that the program on the other end of the connection is trusted. Why is this important? The main attack that this is meant to prevent is the *man-in-the-middle attack*, where the attacker masquerades as both the client and the server to intercept all traffic between them. See figure 13.3 for an overview of this attack.

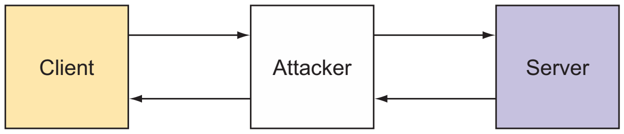

Figure 13.3 Man-in-the-middle attack

As you can see in the figure, a man-in-the-middle attack is exactly what it sounds like:

- A malicious attacker poses as a server, creating a connection with the client, and then poses as the client and creates a connection with the server.
- After that, it can not only decrypt and encrypt all the traffic between the client and server, but it can send arbitrary messages to both the client and the server.

Fortunately, there's hope. The SSL library that MongoDB uses not only provides encryption, but also provides something called *certificate authentication*, which consists of using a trusted third party not involved in the communication to verify the person sending the key is who they claim to be. In theory, the attacker hasn't compromised the third party.

Generating certificates and getting them signed by the trusted third party, known as the certificate authority (CA) for online communication, is outside the scope of this book. There are numerous options here, and each is a topic on its own. To get started, you can work with a CA directly, such as Symantec, Comodo SSL, GlobalSign, and others, or you can use tools that have been built to make this process easier, such as SSLMate.

Once you have a certificate, you can use it in MongoDB like this

```
mongod --clusterAuthMode x509 --sslMode requireSSL --sslPEMKeyFile server.pem
--sslCAFile ca.pem
mongo --ssl --sslPEMKeyFile client.pem
```

where `ca.pem` contains the root certificate chain from the CA and `client.pem` is signed by that CA. The server will use the contents of `ca.pem` to verify that `client.pem` was indeed signed by the CA and is therefore trusted.

Taking these steps will ensure that no malicious program can establish a connection to your database. In the next section, you'll see how to make this more fine-grained and authenticate individual users in a single database.

USER AUTHENTICATION

Although service authentication is great for preventing attackers from even creating a connection to your system, sometimes we want to grant or revoke access on the level of individual users.

NOTE This API changed dramatically from v2.4 to v2.6, and may change again in future versions, so be sure to read the documentation for your server version when you try to set this up. The examples here are for v2.6 and v3.0, which support role-based authentication.

In MongoDB a *role* is essentially a set of privileges, and a *privilege* is any operation that can be performed in MongoDB. The role is a useful concept because sometimes our logical idea of a user "role" doesn't map exactly to primitive database operations. For example, the built-in read role doesn't only allow a user to perform find queries; it also allows users to run certain commands that display statistics about the databases and collections for which they have the read role.

MongoDB has a convenient set of built-in roles, but it also supports user-defined roles if these aren't enough. We won't cover roles in detail here; this topic falls into the category of more advanced and specialized security features, so if you're in a situation where you know you need user-defined roles, we recommend that you consult the most up-to-date documentation for your system and version of MongoDB. For now, we'll jump straight into an example of how to set up basic authentication for a single `mongod`.

SETTING UP BASIC AUTHENTICATION

First, you should start a `mongod` node with `auth` enabled. Note that if this node is in a sharded cluster or a replica set, you also need to pass options to allow it to authenticate with other servers. But for a single node, enabling authentication requires only one flag:

```
$ mongod --auth
```

Now, the first time you connect to the server, you want to add an administrative user account:

```
> use admin
> db.createUser(
  {
    user: "boss",
    pwd: "supersecretpassword",
    roles: [ { role: "userAdminAnyDatabase", db: "admin" } ]
  }
)
```

In our example, we gave this user a role of `userAdminAnyDatabase`, which essentially gives the user complete access to the system, including the ability to add and remove new users, as well as change user privileges. This is essentially the superuser of MongoDB.

Now that we've created our admin user, we can log in as this user:

```
> use admin
> db.auth("boss", "supersecretpassword")
```

We can now create users for individual databases. Once again we use the createUser method. The main differences here are the roles:

```
> use stocks
> db.createUser(
  {
    user: "trader",
    pwd: "youlikemoneytoo",
    roles: [ { role: "readWrite", db: "stocks" } ]
  }
)
> db.createUser(
  {
    user: "read-only-trader",
    pwd: "weshouldtotallyhangout",
    roles: [ { role: "read", db: "stocks" } ]
  }
)
```

Now the trader user has the readWrite role on the stocks database, whereas the read-only-trader only has the read role. This essentially means that the first user can read and write stock data, and the second can only read it. Note that because we added these users to the stocks database, we need to authenticate using that database as well:

```
> use stocks
> db.auth("trader", "youlikemoneytoo")
```

REMOVING A USER

To remove a user, use the dropUser helper on the database it was added to:

```
> use stocks
> db.dropUser("trader")
```

This is a bit heavyweight, so note that you can also revoke user access without completely dropping them from the system using the revokeRolesFromUser helper, and grant them roles again using the grantRolesToUser helper.

To close the session you don't need to explicitly log out; terminating the connection (closing the shell) will accomplish that just fine. But there's a helper for logging out if you need it:

```
> db.logout()
```

Naturally, you can use all the authentication logic we've explored here using the drivers. Check your driver's API for the details.

Localhost exception

You may have noticed we were able to add a user before authenticating our connection. This may seem like a security vulnerability, but it's actually due to a convenience provided by MongoDB called the *localhost exception*.

This means that if the server hasn't yet been configured, any connections from the local machine get full permissions. After you add your first user, unauthenticated connections don't have privileges, as you'd expect. You can pass `--setParameter enableLocalhostAuthBypass=0` on the command line to disable this behavior and set up your first admin user by first starting the server with authentication completely disabled, adding the user, and then restarting with authentication enabled.

This approach isn't any more secure—during the same window where the localhost exception would've been a factor, anyone could've come into your system—but it's another option.

13.4.4 Replica set authentication

Replica sets support the same authentication API just described, but enabling authentication for a replica set requires extra configuration, because not only do clients need to be able to authenticate with the replica set, but replica set nodes also need to be able to authenticate with each other.

Internal replica set authentication can be done via two separate mechanisms:

- Key file authentication
- X509 authentication

In both cases, each replica set node authenticates itself with the others as a special internal user that has enough privileges to make replication work properly.

KEY FILE AUTHENTICATION

The simpler and less secure authentication mechanism is key file authentication. This essentially involves creating a "key file" for each node that contains the password that replica set node will use to authenticate with the other nodes in the replica set. The upside of this is that it's easy to set up, but the downside is that if an attacker compromises just one machine, you'll have to change the password for every node in the cluster, which unfortunately can't be done without downtime.

To start, create the file containing your secret. The contents of the file will serve as the password that each replica set member uses to authenticate with the others. As an example, you might create a file called secret.txt and fill it with the following (don't actually use this password in a real cluster):

```
tOps3cr3tpa55word
```

Place the file on each replica set member's machine and adjust the permissions so that it's accessible only by the owner:

```
sudo chmod 600 /home/mongodb/secret.txt
```

Finally, start each replica set member by specifying the location of the password file using the `--keyFile` option:

```
mongod --keyFile /home/mongodb/secret.txt
```

Authentication will now be enabled for the set. You'll want to create an admin user in advance, as you did in the previous section.

X509 AUTHENTICATION

X509 certificate authentication is built into OpenSSL, the library MongoDB uses to encrypt network traffic. As we mentioned earlier, obtaining signed certificates is outside the scope of this book. However, once you have them, you can start each node like this

```
mongod --replSet myReplSet --sslMode requireSSL --clusterAuthMode x509 --
sslClusterFile --sslPEMKeyFile server.pem --sslCAFile ca.pem
```

where `server.pem` is a key signed by the certificate authority that `ca.pem` corresponds to.

There's a way to upgrade a system using key file authentication to use X509 certificates with no downtime. See the MongoDB docs for the details on how to do this, or check in the latest MMS documentation to see whether support has been added to MMS automation.

13.4.5 Sharding authentication

Sharding authentication is an extension of replica set authentication. Each replica set in the cluster is secured as described in the previous section. In addition, all the config servers and every `mongos` instance can be set up to authenticate with the rest of the cluster in exactly the same way, using either a shared key file or using X509 certificate authentication. Once you've done this, the whole cluster can use authentication.

13.4.6 Enterprise security features

Some security features exist only in MongoDB's paid enterprise plug-in. For example, the authentication and authorization mechanisms that allow MongoDB to interact with Kerberos and LDAP are enterprise. In addition, the enterprise module adds auditing support so that security-related events get tracked and logged. The MongoDB docs will explicitly mention if a particular feature is enterprise only.

13.5 Administrative tasks

In this section, we'll cover some basic administrative tasks, including importing and exporting data, dealing with disk fragmentation, and upgrading your system.

13.5.1 Data imports and exports

If you're migrating an existing system to MongoDB, or if you need to seed the database with information from, something like a data warehouse, you'll need an efficient import method. You might also need a good export strategy, because you may have to export data from MongoDB to external processing jobs. For example, exporting data to Hadoop for batch processing has become a common practice.[24]

There are two ways to import and export data with MongoDB:

- Use the included tools, `mongoimport` and `mongoexport`.
- Write a simple program using one of the drivers.[25]

MONGOIMPORT AND MONGOEXPORT

Bundled with MongoDB are two utilities for importing and exporting data: `mongoimport` and `mongoexport`. You can use `mongoimport` to import JSON, CSV, and TSV files. This is frequently useful for loading data from relational databases into MongoDB:

```
$ mongoimport -d stocks -c values --type csv --headerline stocks.csv
```

In the example, you import a CSV file called stocks.csv into the `values` collection of the `stocks` database. The `--headerline` flag indicates that the first line of the CSV contains the field names. You can see all the import options by running `mongoimport --help`.

Use `mongoexport` to export all of a collection's data to a JSON or CSV file:

```
$ mongoexport -d stocks -c values -o stocks.csv
```

This command exports data to the file stocks.csv. As with its counterpart, you can see the rest of `mongoexport`'s command options by starting it with the `--help` flag.

CUSTOM IMPORT AND EXPORT SCRIPTS

You're likely to use MongoDB's import and export tools when the data you're dealing with is relatively flat; once you introduce subdocuments and arrays, the CSV format becomes awkward because it's not designed to represent nested data.

When you need to export a rich document to CSV or import a CSV to a rich MongoDB document, it may be easier to build a custom tool instead. You can do this using any of the drivers. For example, MongoDB users commonly write scripts that connect to a relational database and then combine the data from two tables into a single collection.

That's the tricky part about moving data in and out of MongoDB: the way the data is modeled may differ between systems. In these cases, be prepared to use the drivers as your conversion tools.

13.5.2 Compaction and repair

MongoDB includes a built-in tool for repairing a database. You can initiate it from the command line to repair all databases on the server:

```
$ mongod --repair
```

[24] There's also a Hadoop plug-in for MongoDB, sometimes known as the MongoDB Connector for Hadoop.
[25] You can also use a tool known as mongoconnector to keep a different storage system in sync with MongoDB.

Or you can run the `repairDatabase` command to repair a single database:

```
> use cloud-docs
> db.runCommand({repairDatabase: 1})
```

Repair is an offline operation. While it's running, the database will be locked against reads and writes. The repair process works by reading and rewriting all data files, discarding any corrupted documents in the process. It also rebuilds each index. This means that to repair a database, you need enough free disk space to store the rewrite of its data. To say repairs are expensive is an understatement, as repairing a large database can take days and impact traffic to other databases on the same node.

MongoDB's repair was originally used as a kind of last-ditch effort for recovering a corrupted database. In the event of an unclean shutdown, without journaling enabled, a repair is the only way to return the data files to a consistent state, and even then you may lose data. Fortunately, if you deploy with replication, run at least one server with journaling enabled, and perform regular off-site backups, you should never have to recover by running a repair. Relying on repair for recovery is foolish. Avoid it.

What, then, might a database repair be good for? Running a repair will compact the data files and rebuild the indexes. As of the v2.0 release, MongoDB doesn't have great support for data file compaction. If you perform lots of random deletes, and especially if you're deleting small documents (less than 4 KB), it's possible for total storage size to remain constant or grow despite these regularly occurring deletes. Compacting the data files is a good remedy for this excess use of space.[26]

If you don't have the time or resources to run a complete repair, you have two options, both of which operate on a single collection:

- Rebuilding indexes
- Compacting the collection

To rebuild indexes, use the `reIndex()` method:

```
> use cloud-docs
> db.spreadsheets.reIndex()
```

This might be useful, but generally speaking, index space is efficiently reused. The data file space is what can be a problem, so the `compact` command is usually a better choice.

`compact` will rewrite the data files and rebuild all indexes for one collection. Here's how you run it from the shell:

```
> db.runCommand({ compact: "spreadsheets" })
```

This command has been designed to be run on a live secondary, obviating the need for downtime. Once you've finished compacting all the secondaries in a replica set, you can step down the primary and then compact that node. If you must run the compact

[26] Compacting your data on disk may also result in more efficient use of space in RAM.

command on the primary, you can do so by adding `{force: true}` to the command object. Note that if you go this route, the command will write lock the system:

```
> db.runCommand({ compact: "spreadsheets", force: true })
```

On WiredTiger databases, the `compact()` command will release unneeded disk space to the operating system. Also note that the `paddingFactor` field, which is applicable for the MMAPv1 storage engine, has no effect when used with the WiredTiger storage engine.

13.5.3 Upgrading

As with any software project, you should keep your MongoDB deployment as up to date as possible, because newer versions contain many important bug fixes and improvements.

One of the core design principles behind MongoDB is to always ensure an upgrade is possible with no downtime. For a replica set, this means a rolling upgrade, and for a sharded cluster, this means that `mongos` routers can still function against mixed clusters.

You can take one of two paths when dealing with upgrades:

- First, you can choose to do the process manually. In this case you should check the release notes describing the upgrade process and read them carefully. Sometimes there are important caveats or steps that must be taken to ensure that the upgrade is safe. The advantage here is that you have complete control over the process and know exactly what's happening when.
- The second option is to again use the MongoDB Management Service. MMS Automation not only can be used to provision your nodes, but can also be used to upgrade them. Be sure to read the release notes for the version you're upgrading to as well as the MMS documentation to be sure you know what's happening under the hood.

13.6 Performance troubleshooting

Performance in MongoDB is a complicated issue that touches on nearly every other topic. It's nearly impossible to know exactly how your application will perform before you test it with your specific workload and environment. A node on the Joyent Cloud will have completely different performance characteristics than a node on EC2, which will again have completely different performance characteristics than a node that you installed in your own private datacenter.

In this section, we'll address some common issues that come up in MongoDB support, as least up through version 3.0, that are important to watch out for. First, we'll reintroduce the working set concept; then we'll cover two ways the working set can affect your system: the performance cliff and query interactions.

In the end, it's up to you to monitor your system closely and understand how it's behaving. Good monitoring could be the difference between quickly resolving, or even predicting and preventing performance issues, and having angry users be the first to tell you your systems have slowed to a crawl.

13.6.1 Working set

We've covered the idea of the working set in various parts of this book, but we'll define it here again with a focus on your production deployment.

Imagine you have a machine with 8 GB of RAM, running a database with an on-disk size of 16 GB, not including indexes. Your *working set* is how much data you're accessing in a specified time interval. In this example, if your queries are all full collection scans, your "working set" will be 16 GB because to answer those queries your entire database must be paged into memory.

But if your queries are properly indexed, and you're only querying for the most recent quarter of the data, most of your database can stay on disk, and only the 2 GB that you need, plus some extra space for indexes, needs to be in memory.

Figure 13.4 shows visually what this means for your disk use.

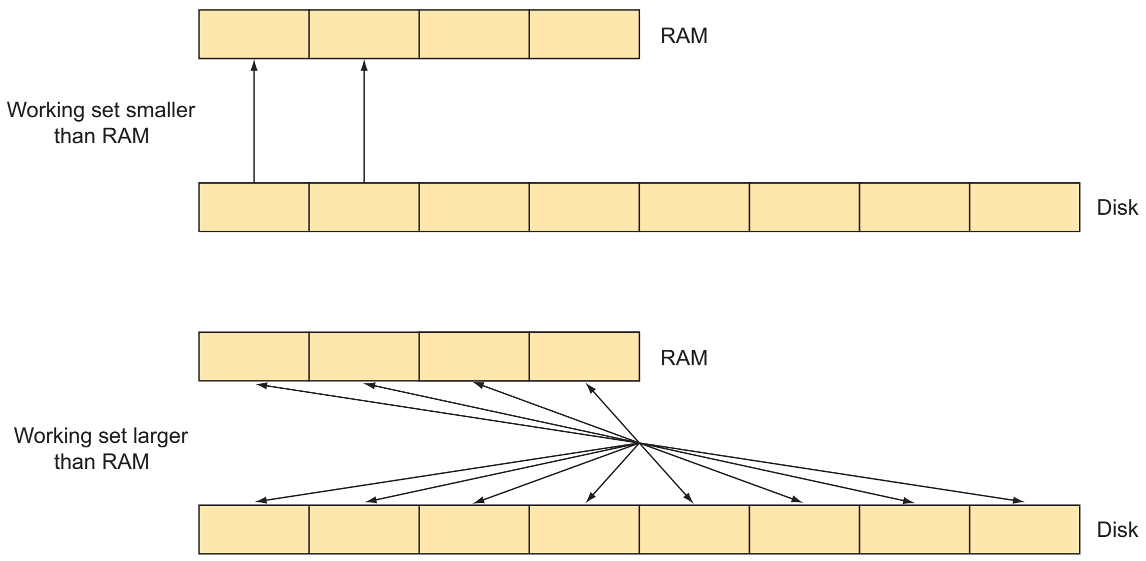

Figure 13.4 The impact on disk usage when the working set exceeds RAM

In the bottom row of the figure, when you're doing full table scans, you have a lot of *thrashing*, or moving blocks of data in and out of RAM, because you can't keep all 16 GB in your 8 GB of RAM at once.

In the top example, you can bring the 2 GB you need into memory, and then keep it there, answering all the queries with only minimal disk access. This illustrates not only why keeping your working set in RAM is extremely important, but also shows how a simple change, such as adding the right index, can completely change your performance characteristics.

13.6.2 Performance cliff

Closely related to the working set concept is an unfortunate artifact of the way MongoDB runs queries. As of 3.0, MongoDB doesn't limit the resources a single operation can use[27], nor does it explicitly push back on clients when it's starting to become overwhelmed. As long as you aren't approaching MongoDB's limits, this won't be a problem, but as you reach the limit, you may see a "performance cliff."

To see why this is the case, imagine you have a system with 8 GB of RAM, running a database that's 64 GB on disk. Most of the time, your working set is only 6 GB, and your application performs perfectly well. Now, fast-forward to Christmas Eve, when you see a traffic spike big enough to bring you to 8 GB. MongoDB will start to slow down, because the working set no longer fits in memory and will start thrashing to disk. Despite this, MongoDB will continue to accept new requests.

The combination of slowing down while still accepting new requests is disastrous for performance, and you may see a steep drop-off at this point. If you think this will be a concern for you, we strongly recommend that you understand exactly how much load your MongoDB deployment can handle, and if possible put some kind of load balancer in front of it that will ensure that you steer away from the edge of this precipice.

13.6.3 Query interactions

Another side effect of the fact that MongoDB doesn't enforce resource limits is that one badly behaving query can affect the performance of all other queries on the system.

It's the same drill as before. Assume you have a working set of 2 GB, with a 64 GB database. Everything may be going well, until someone runs a query that performs a full collection scan. This query will not only place a huge amount of load on the disk, but may also page out the data that was being used for the other queries on the system, causing slowdown there as well. Figure 13.4 from earlier illustrates this issue, where the top represents normal query load and the bottom represents the load after the bad query.

This is actually another reason why access controls are important. Even if you get everything else right, one table scan by an intern can hose your system. Make sure everyone who has the ability to query your database understands the consequences of a bad query.[28]

---

[27] A crude approximation is the `maxTimeMS` cursor option, which sets a maximum time limit for processing operations. This won't prevent one resource-hungry operation from interfering with others, but it will at least kill operations that run for longer than expected.

[28] You can mitigate this a bit by running your long-running queries on secondaries, but remember from chapter 11 that the secondary must have enough bandwidth to keep up with the primary, otherwise it will fall off the end of the oplog.

All about the indexes

When you discover a performance issue, indexes are the first place you should look. Unless your operation is insert only, indexes are a vital part of ensuring good performance.

Chapter 8 outlines a procedure for identifying and fixing slow operations that involves enabling the query profiler and then ensuring that every query and update uses an index efficiently. In general, this means each operation scans as few documents as possible.

It's also important to make sure there are no redundant indexes, because a redundant index will take up space on disk, require more RAM, and demand more work on each write. Chapter 8 mentions ways to eliminate these redundant indexes.

What then? After auditing your indexes and queries, you may discover inefficiencies that, when corrected, fix the performance problems altogether. You'll no longer see slow query warnings in the logs, and the `iostat` output will show reduced utilization. Adjusting indexes fixes performance issues more often than you might think; this should always be the first place you look when addressing a performance issue.

13.6.4 Seek professional assistance

The sources of performance degradations are manifold and frequently idiosyncratic. Anything from poor schema design to sneaky server bugs can negatively affect performance. If you think you've tried every possible remedy and still can't get results, consider allowing someone experienced in the ways of MongoDB to audit your system. A book can take you far, but an experienced human being can make all the difference in the world. When you're at a loss for ideas and in doubt, seek professional assistance. The solutions to performance issues are sometimes entirely unintuitive.

When or if you seek help, be sure to provide all the information you have about your system when the issue occurred. This is when the monitoring will pay off. The official standard used by MongoDB is MMS Monitoring, so if you're using MongoDB support, being set up with MMS Monitoring will speed up the process significantly.

13.7 Deployment checklist

We've covered a lot of topics in this chapter. It may seem overwhelming at first, but as long as you have the main areas covered, your system will keep running smoothly. This section is a quick reference for making sure you've got the important points covered:

- Hardware
  - *RAM*—Enough to handle the expected working set.
  - *Disk space*—Enough space to handle all your data, indexes, and MongoDB internal metadata.
  - *Disk speed*—Enough to satisfy your latency and throughput requirements. Consider this in conjunction with RAM—less RAM usually means more disk usage.

  - *CPU*—Usually not the bottleneck for MongoDB, but if you're getting low disk utilization but low throughput, you may have a CPU bound workload. Check this as part of careful performance testing.
  - *Network*—Make sure the network is fast and reliable enough to satisfy your performance requirements. MongoDB nodes communicate with each other internally, so be sure to test every connection, not just the ones from your clients to the `mongos` or `mongod` servers.
- Security
  - *Protection of network traffic*—Either run in a completely isolated environment or make sure your traffic is encrypted using MongoDB's built-in SSL support or some external method such as a VPN to prevent man-in-the-middle attacks.
  - *Access control*—Make sure only trusted users and clients programs can operate on the database. Make sure your interns don't have the "root" privilege.
- Monitoring
  - *Hardware usage (disks, CPU, network, RAM)*—Make sure you have some kind of monitoring setup for all your hardware resources that will not only keep track of the usage, but also alert you if it goes above a certain threshold.
  - *Health checks*—Periodically make sure your servers are up and responsive, and will alert you if anyone stops calling back.
  - *MMS Monitoring*—Monitor your services using MMS Monitoring. Not only does this provide monitoring, health checks, and alerts, but it's what the MongoDB support team will use to help you if you run into trouble. Historically it's been free to use, so don't hesitate to add this to your deployment.
  - *Client performance monitoring*—Periodically run automated end-to-end tests as a client to ensure that you're still performing as well as you expect. The last thing you want is for a client to be the first one to tell you that your application is slow.
- Disaster recovery
  - *Evaluate risk*—Imagine that you've lost all your data. How sad do you feel? In all seriousness, losing your data may be worse in some applications than others. If you're analyzing Twitter trends, losing your data may cost a week's worth of time, whereas if you're storing bank data, losing that may cost quite a bit more. When you do this evaluation, assume that a disaster of some kind will happen, and plan accordingly.
  - *Have a plan*—Create a concrete plan for how you'll recover in each failure case. Depending on how your system fails, you may react completely differently.
  - *Test your plan*—Be sure to test your plan. The biggest mistake people make with backups and disaster recovery is assuming that having a backup or a plan is enough. It's not enough. Maybe the backup is getting corrupted. Maybe it's in a format that's impossible to reimport into your production systems. As in a production system, many things can go wrong, so it's important to make sure your recovery strategy works.

  - *Have a backup plan*—Your first disaster recovery plan might fail. When it does, have a last resort available. This doesn't have to be an appealing option, but you'll be happy it's there if you get desperate.
- Performance
  - *Load testing*—Make sure you load test your application with a realistic workload. In the end, this is the only way to be sure that your performance is what you expect.

13.8 Summary

This chapter has presented the most important considerations for deploying MongoDB in production. You should have the knowledge you need to select the right hardware for MongoDB, monitor your deployments, and maintain regular backups. In addition, you should have some idea about how to go about resolving performance issues. Ultimately, this knowledge will develop with experience. But MongoDB is predictable enough to be amenable to the simple heuristic presented here—except for when it isn't. MongoDB tries to make life simple, but databases and their interactions with live applications are frankly complex. The guidelines in this chapter can point you in the right direction, but in the end it's up to you to take an active role in understanding your system. Make sure you take full advantage of the resources available, from the MongoDB documentation all the way to the official and community support from MongoDB maintainers and users.