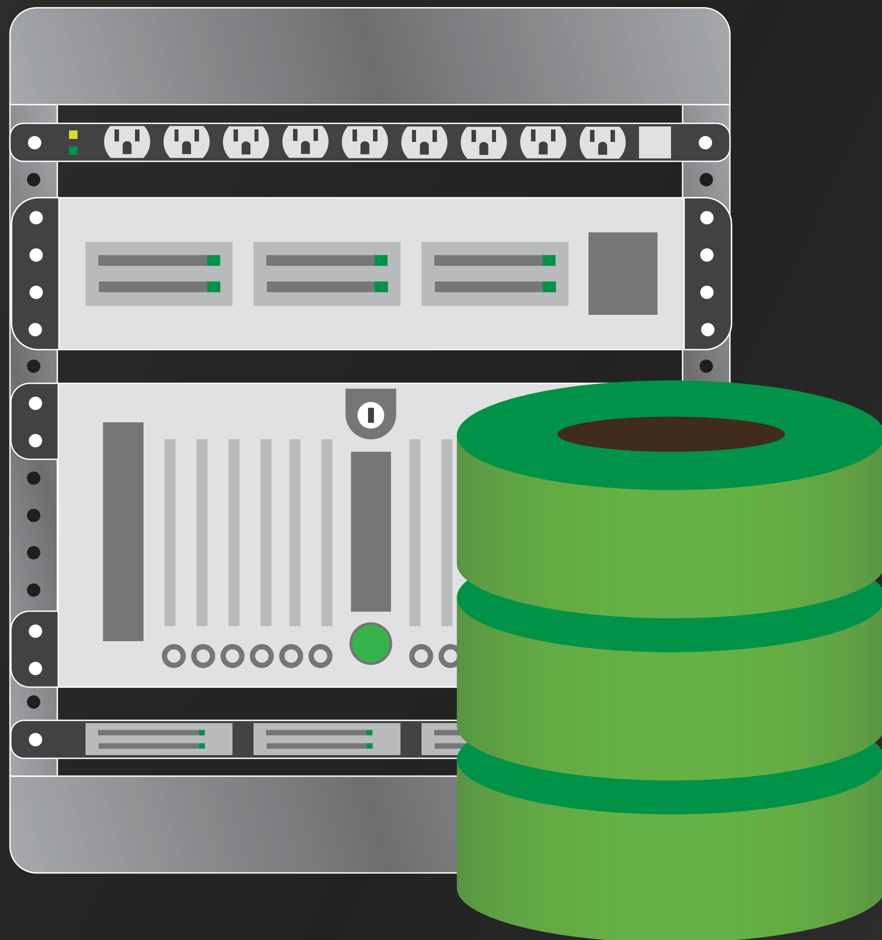
severalnines

Become a MongoDB DBA: Bringing MongoDB to Production



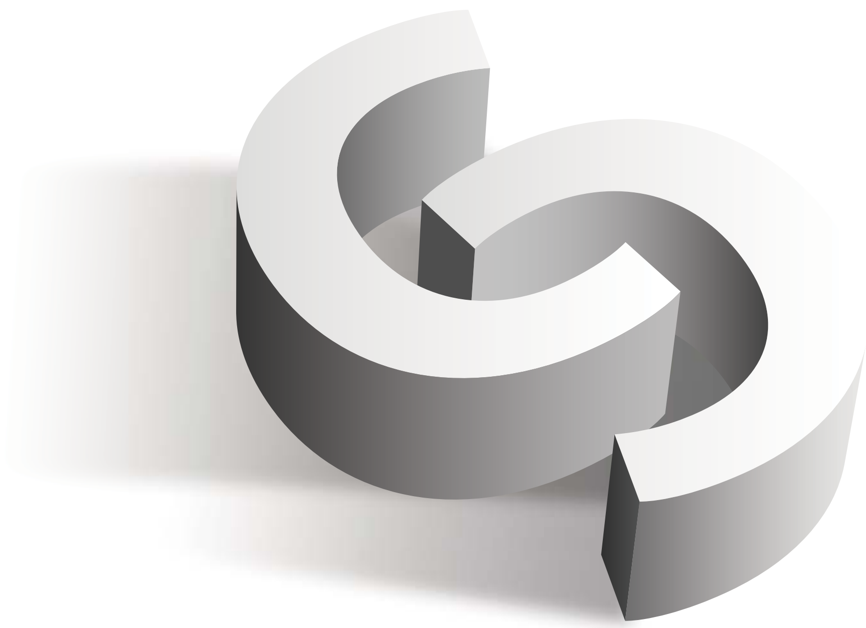


Table of Contents

1. Introduction 4

2. Choosing the right MongoDB version 5
- 2.1. Topologies 5
- 2.2. Storage engines 6

3. Securing MongoDB 8
- 3.1. Enable authentication 8
- 3.2. Authorize users by roles 8
- 3.3. Add a replication keyfile 9
- 3.4. Does your application require public access? 9
- 3.5. Enable firewall rules or security groups 10
- 3.6. Enabling SSL 11

4. Monitoring and Trending 12
- 4.1. Host metrics 12
- 4.2. dbStats metrics 12
- 4.3. serverStatus metrics 13
- 4.4. Oplog metrics 14
- 4.5. MongoDB locks 16
- 4.6. WiredTiger metrics 18
  - 4.6.1. Locks and concurrency 18
  - 4.6.2. Transactions 18

5. Backup and Recovery 20
- 5.1. Logical backups 20
- 5.2. Physical backups 20
- 5.3. Sharded MongoDB backups 21
- 5.4. Backup strategies 21
  - 5.4.1. Offsite backups 21
  - 5.4.2. Backup encryption 22
  - 5.4.3. Recovery 22

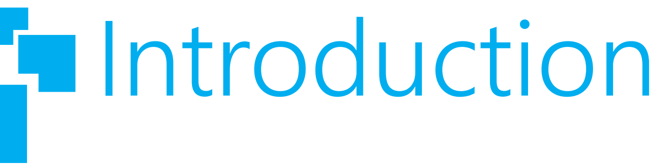
6. Scaling MongoDB 23
- 6.1. Read scaling 23
- 6.2. Reading from a secondary 24
  - 6.2.1. Setting read preference 25
  - 6.2.2. Reading from a secondary in a shard 26
- 6.3. MongoDB write scaling (sharding) 26
  - 6.3.1. Sharding tier 26
  - 6.3.2. Shard tier 27
  - 6.3.3. Managing shards 28

7. Conclusion 29

8. About ClusterControl 30

9. About Severalnines 30

10. Related Resources from Severalnines 31



Introduction

MongoDB is a document data store that has been around for almost a decade already. During the past five years MongoDB evolved into a mature product that features enterprise grade features like scalability (sharding), security and resilience. Not all of these features in the MongoDB Enterprise version are generally available for the public. However with the recent improvements made on the Percona version of MongoDB, some of these feature are now available. With the addition of third party components and storage engines, like RocksDB, the ecosystem of MongoDB has become a flourishing one.

In this whitepaper we will shed a light on choosing the right version for your cluster and cover the basics of preparing MongoDB for production.

Choosing the right MongoDB version

There are quite a few versions, implementations and topologies to choose from. MongoDB itself is, like MySQL, an open source platform that anyone can use and improve. This means that various parties have created their own versions or additions to it. A brief overview could be made like this:

	Vendor	Open source	Notes	Cost
MongoDB Community	MongoDB	yes		free
MongoDB Enterprise	MongoDB	no		paid
MongoDB Atlas	MongoDB	no	Enterprise cloud solution	paid
Percona Server for MongoDB	Percona	yes	Additional enterprise features	free

As you can see in the table above, MongoDB is the dominant vendor, offering both the community and enterprise versions. Percona enriches the MongoDB community version with some enterprise grade features like audit logging and the memory engine, that are also present in the MongoDB enterprise version. The two versions are really comparable with each other.

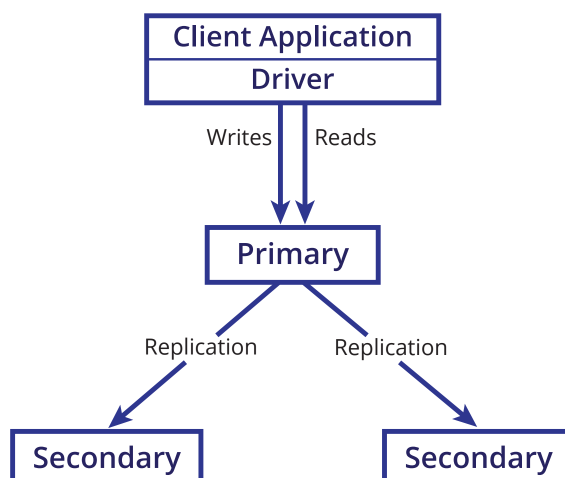
Apart from these options for bare bones infrastructures, there are many cloud vendors that offer pre-installed MongoDB database hosts and most of these reside on large cloud providers like AWS, Google and Azure. These cloud instances are in no way comparable to MongoDB Atlas, which offers a complete solution from the cloud interface. MongoDB Atlas not only offers deployment of your nodes in AWS, but also snapshots of your data stored in Amazon S3. Alternatives to the MongoDB Atlas service are Rackspace ObjectRocket and Severalnines NinesControl.

2.1. Topologies

After you have chosen the right vendor and version, it is also essential to choose your topology before deployment. Depending on your topology you need to alter or change your deployment and configuration strategy. MongoDB basically supports three major types of toplogies: standalone, replicaSet and sharded.

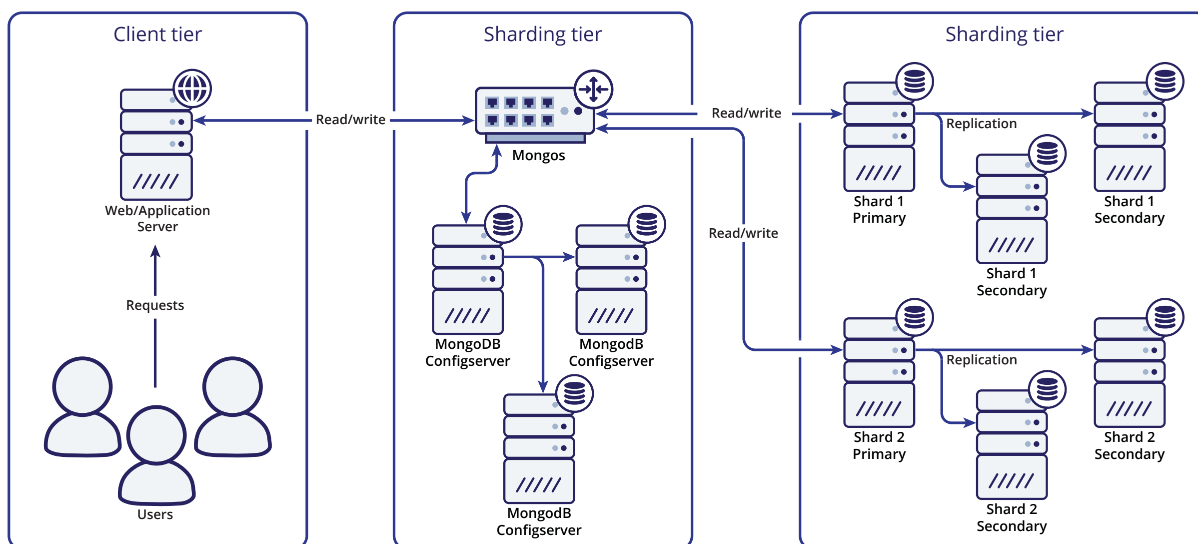
We can simply start with the single standalone MongoDB instance. This is comparable to the MySQL single instance and naturally this topology will not have any data replicated from one host to another.

Once we replicate data between nodes, this is called a ReplicaSet in MongoDB. MongoDB will ensure transactions will be written to the oplog (comparable with the MySQL binlog) in more than just the primary (master): also to secondary nodes (slaves). You can configure the transaction to be confirmed by the primary after writing to it.



For a ReplicaSet we need at least two instances (one primary and one secondary) to confirm a write, but it is advisable to use at least three. A MongoDB ReplicaSet can best be compared to a hybrid between traditional MySQL replication and Galera synchronous replication.

The other topology to mention is MongoDB Sharding. MongoDB will be able to shard based upon the data stored in the Config Servers, and route the queries to the correct shards. In the picture below, the Config Servers and Shards are all independent ReplicaSets.



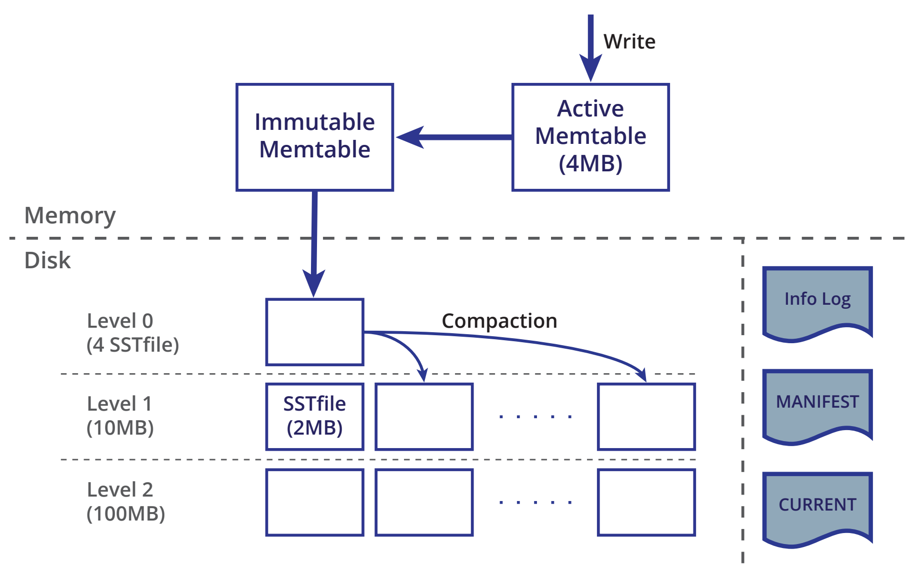
2.2. Storage engines

Just as important as choosing your topology, is the choice of storage engine. MongoDBs default storage engine used to be MMap up until version 3.0. MMap is comparable to MyISAM in structure: it is a file based storage engine that (heavily) relies on filesystem caching. The benefit is the low overhead of the storage engine, but just as with MyISAM, locking is performed on collection level (or table level in MySQL). This is due to the file based storage system, where the writing thread holds the lock on the collection. This means that any MongoDB primary that receives a heavy write

workload on one schema or collection is bound have contention issues. The structure itself is B-tree based and as it is appending inserted data to the file, it is generally over allocating space on disk.

The default storage engine as of version 3.0 is WiredTiger. This storage engine has overcome the biggest shortcoming of MMap: it features document level locking. This means multiple clients can write data to the same collection now, as long as they don't attempt to write to the same document. WiredTiger is, just like MMap, B-tree based but is able to support LSM-trees. It supports compression and index prefix compression as well, and this performs reasonably well due to the choice of using Snappy as default compression. WiredTiger is slower in pure performance than MMap, but thanks to its document locking in production workloads, it is often faster. We would say WiredTiger is a good all round storage engine.

RocksDB has been developed by Facebook to solve flash based storage issues. As SSDs behave differently to traditional (spinning) harddisks, the approach of a high performance datastore had to be different. On flash storage a read is a very inexpensive and fast operation as a whole cell can be read instantly, while a single write operation is considered an expensive operation as it needs to read, erase, modify and then write the data back to a new place in the storage. This means flash based systems will perform less good when writing small pieces of random data. Because of this nature, RocksDB is an append-only storage system that uses a Log Structured Merge tree (LSM-tree) to store the data in various layers and merges these data sets to a lower level to maintain the full dataset. The benefit of this storage engine is that it has excellent insert, update and delete performance. It also performs really well on time-bound data (last 50 messages in your inbox, range queries) but has poor performance on queries that select on other criteria.



Another storage engine would be the Percona TokuDB engine. This engine is based upon fractal trees and should give a much higher insert and update performance than WiredTiger. Percona recently stopped development on this storage engine, so we would advise against using this engine in production.

Also, unless you do not need massive insert performance and have fairly common query patterns, we would recommend to stick to the WiredTiger storage engine as it delivers reasonable performance and is tunable.

Securing MongoDB

MongoDB comes with very little security out of the box: for instance, authorization is disabled by default. In other words: *by default anyone has root rights over any database*. One of the changes MongoDB applied to mitigate risks was to change its default binding to 127.0.0.1. This prevents it from being bound to the external ip address, but naturally this will be reverted by most people who install it.

Recently unsecured MongoDB instances have been subject to ransomware. This ransomware scans for unprotected MongoDB instances that are open to the internet and accept any incoming connection. Once the attackers have taken control over the MongoDB instance, most of them hijack the data by copying it onto their own storage. After making a copy they will erase the data on the server, and leave a database with a single collection demanding ransom for the data. In addition, some also threaten to erase the backup that they hold hostage if they don't get paid within 3 days. Some victims, who allegedly paid, in the end never received their backup.

As you can see, securing your MongoDB instance is just as vital as securing any other database! You can secure MongoDB by simply enabling authentication and authorization, only allow specific hosts to connect and use SSL.

3.1. Enable authentication

This is the easiest solution to keep unwanted people outside: simply enable authentication in MongoDB. To explicitly do this, you will need to put the following lines in the mongod.conf:

```
1   security:
2           Authentication: on
```

If you have set the replication keyfile in the mongod.conf, you will also implicitly enable authentication.

3.2. Authorize users by roles

Even if you have enabled authentication, don't just give every user an administrative role. This would be very convenient from the user perspective as they can literally perform every task thinkable, and would not depend on an administrator to execute a task on their behalf. But for any attacker, this is just as convenient: as soon as they have access to one single account, they also immediately have the administrative role.

MongoDB has a diversity of roles, and for any type of task an applicable role is present. Ensure that the user carrying the administrative role is a user that isn't part of the application stack. This should slim down the chances of an account breach to result into disaster.

When provisioning MongoDB from ClusterControl, we deploy new MongoDB replicaSets and sharded clusters with a separate admin and backup user.

3.3. Add a replication keyfile

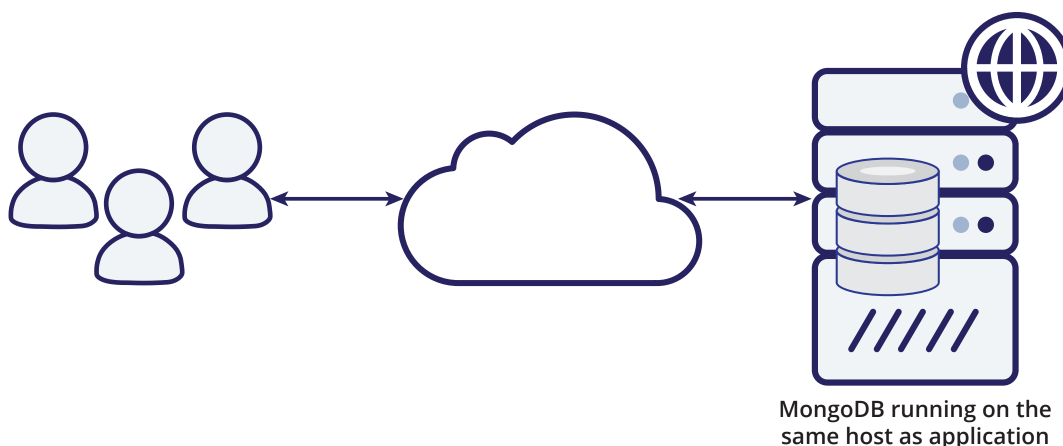
As mentioned earlier, enabling the replication keyfile will implicitly enable authentication in MongoDB. But there is a much more important reason to add a replication keyfile: once added, only hosts with the file installed are able to join the replicaSet.

Why is this important? Adding new secondaries to a replicaSet normally requires the ***clusterManager*** role in MongoDB. Without authentication, any user can add a new host to the cluster and replicate your data over the internet. This way an attacker could silently and continuously tap into your data.

With the keyfile enabled, the authentication of the replication stream will be encrypted. This ensures nobody can spoof the ip of an existing host, and pretend to be another secondary that isn't supposed to be part of the cluster. In ClusterControl, we deploy all MongoDB replicaSets and sharded clusters with a replication keyfile.

3.4. Does your application require public access?

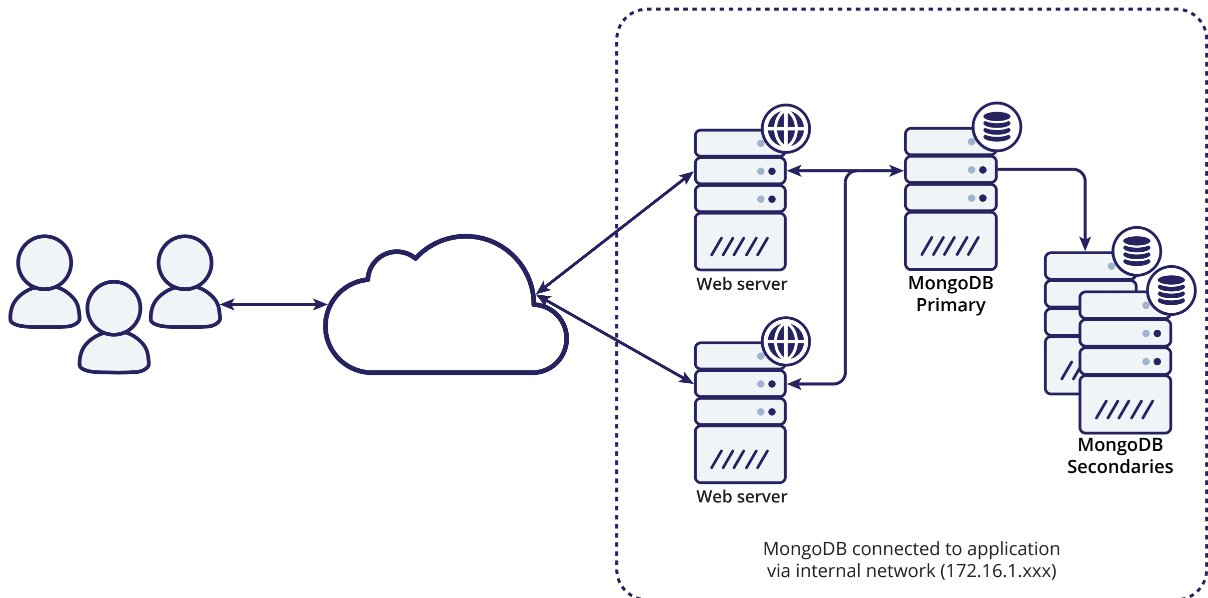
If you have enabled MongoDB to be bound on all interfaces, you may want to review if your application actually needs external access to the datastore. If your application is a single hosted solution and resides on the same host as the MongoDB server, it can suffice by binding MongoDB to localhost.



This requires the following line to be added/changed in the mongod.conf and a restart is required:

```
1  net:
2    bindIp: 127.0.0.1
```

In many hosting and cloud environments with multi-tenant architectures, applications are put on different hosts than where the datastore resides. The application then connects to the datastore via the private (internal) network. If this applies to your environment, you need to ensure to bind MongoDB only to the private network.



This requires the following line to be added/changed in the mongod.conf and a restart is required:

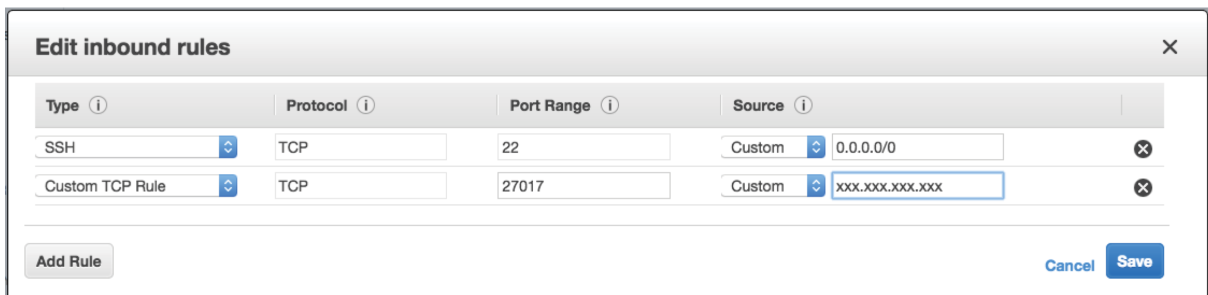
```
1  net:
2    bindIp: 127.0.0.1,172.16.1.234
```

3.5. Enable firewall rules or security groups

It is a good practice to enable firewall rules on your hosts, or security groups with cloud hosting. Simply disallowing the MongoDB port ranges from outside will keep most attackers outside.

There would still be another way to get in: from the inside. If the attacker would gain access to another host in your private (internal) network, they still could access your datastore. A good example would be proxying tcp/ip requests via a http server. Add firewall rules to the MongoDB instance and deny any other host except the hosts that you know for sure need access. This should, at least, limit the number of hosts that could potentially be used to get your data. And as indicated in Tip 1: enable authentication, even if someone proxies into your private network they can't steal your data.

Also, if your application does require MongoDB to be available on the public interface, you can limit the hosts accessing the database by simply adding similar firewall rules.



3.6. Enabling SSL

Enabling encryption on database communication has become a necessity over the past few years, especially when databases are deployed in the cloud. This accounts for both internal and external traffic. MongoDB supports encryption of both client-server connection and intra-cluster communication.

Once you enable Transport Encryption in MongoDB, all of the network traffic of MongoDB will be encrypted using TLS/SSL (Transport Layer Security/Secure Sockets Layer). When enabled, both internal and external communication will be encrypted. There is no possibility to do only one of them.

You can configure MongoDB to use SSL by enabling it and adding the certificate file:

```
1    net:
2       ssl:
3          mode: requireSSL
4          PEMKeyFile: /etc/ssl/mongodb.pem
```

Monitoring and Trending

To manage your databases, you as the DB admin would need good visibility into what is going on. Remember that if a database is not available or not performing, you will be the one under pressure so you want to know what is going on. If there is no monitoring and trending system available, this should be the highest priority. In this chapter we will summarize the most important metrics to keep an eye out for and why you should monitor those.

4.1. Host metrics

Host metrics are equally important to MongoDB as they are for MySQL or any other database. MongoDB is a database system, so it will behave in a large degree the same as MySQL. High load, low IO and low CPU utilization? Your MySQL instinct will be right here as well: there must be some sort of locking issue.

So in terms of host metrics, capture everything you would normally do for any other database:

- CPU usage / load / cpusteal
- Memory usage
- IO
- Network

4.2. dbStats metrics

The most basic check you wish to perform on any MongoDB host is whether the service is running and responding. But whether the service is up or down is not enough. Alongside that check, you can fetch the database statistics to give you the most basic metrics

```
1   my_mongodb_0:PRIMARY> use admin
2   switched to db admin
3   my_mongodb_0:PRIMARY> db.runCommand( { dbStats : 1 } )
4   {
5           "db" : "admin",
6           "collections" : 2,
7           "objects" : 2,
8           "avgObjSize" : 198,
9           "dataSize" : 396,
10          "storageSize" : 32768,
11          "numExtents" : 0,
12          "indexes" : 3,
13          "indexSize" : 49152,
14          "ok" : 1
15  }
```

It is important to switch to the admin database, as otherwise you will capture the stats from the database that you are using. We can spot already here a couple of important stats: the number of collections (like tables), objects, data/storage size and index size. This is a good metric to keep an eye on the growth rate of your MongoDB cluster.

4.3. serverStatus metrics

The server status is comparable to the MySQL ***show global status*** command: it will contain the most important stats from MongoDB. Depending on the storage engine that you are using, this will contain the stats for WiredTiger, MongoRocks or TokuMX.

For example, if we wish to only see the *replicaSet* information we have to filter out the remainder:

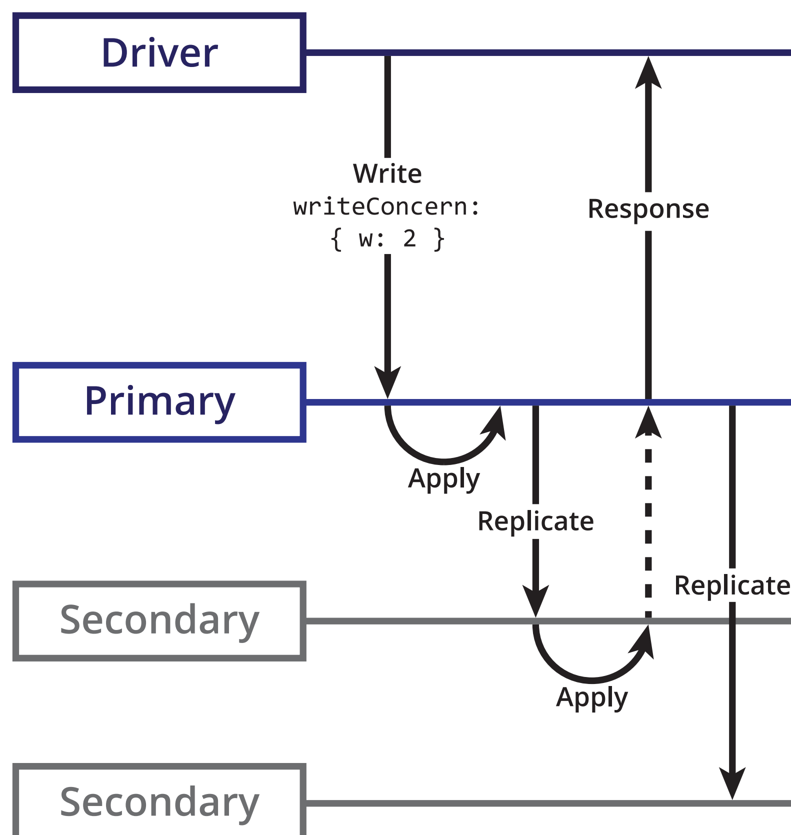
```
1    my_mongodb_0:PRIMARY> db.serverStatus({ wiredTiger:
     0, asserts: 0, metrics:  0, tcmalloc: 0, locks: 0, op-
     countersRepl: 0, opcounters: 0, network: 0, globalLock: 0,
     extra_info: 0, connections: 0, storageEngine: 0})
2    {
3            "host" : "n2",
4            "advisoryHostFQDNs" : [ ],
5            "version" : "3.2.6-1.0",
6            "process" : "mongod",
7            "pid" : NumberLong(12122),
8            "uptime" : 600,
9            "uptimeMillis" : NumberLong(599289),
10           "uptimeEstimate" : 576,
11           "localTime" : ISODate("2016-06-18T11:13:09.080Z"),
12           "repl" : {
13                   "hosts" : [
14                           "10.10.32.11:27017",
15                           "10.10.32.12:27017",
16                           "10.10.32.13:27017"
17                   ],
18                   "setName" : "my_mongodb_0",
19                   "setVersion" : 1,
20                   "ismaster" : true,
21                   "secondary" : false,
22                   "primary" : "10.10.32.11:27017",
23                   "me" : "10.10.32.11:27017",
24                   "electionId" : ObjectId("7fffff
     ff0000000000000001"),
25                   "rbid" : 1522923277
26           },
27           "ok" : 1
28   }
```

As you can see the flexibility of JSON comes in handy here. Unlike MySQL you are not bound by a predefined set of status variables. When enabled, the *wiredTiger* object

is present here, while when using MongoRocks, we would have an additional *rocksdb* object. You can find the storage engine in use under the *storageEngine* object.

4.4. Oplog metrics

With MongoDB replication, the most important aspect is the oplog. As we described in a blog post about MongoDB configuration, the oplog is comparable to the MySQL binary log. It keeps a history of all transactions in this log file, but contrary to the MySQL binary log, MongoDB only has one single collection where it stores them. This collection is limited in size. This means that once it is full, old transactions will get purged as new transactions come in. It will evict the oldest transaction first, so the method used here is FIFO (First In, First Out). Therefore the most important metric to watch is *the replication window*: the duration of transactions kept in the oplog.



Why is this important? Suppose one of the secondary nodes loses network connectivity with the primary, it will no longer replicate data from the primary. Once it comes back online this secondary node has to catch up with already replicated transactions. It will use the oplog for this purpose. If the secondary node went offline for too long, it can't use the oplog anymore and a full sync is necessary. A full sync, just like the SST in Galera, is an expensive operation and you would want to avoid this.

```
1   mongo_replica_0:PRIMARY> db.getReplicationInfo()
2   {
3           “logSizeMB” : 1895.7751951217651,
4           “usedMB” : 0.01,
5           “timeDiff” : 11,
6           “timeDiffHours” : 0,
```

```
7          "tFirst" : "Fri Jul 08 2016 10:56:01 GMT+0000
   (UTC)",
8          "tLast" : "Fri Jul 08 2016 10:56:12 GMT+0000 (UTC)",
9          "now" : "Fri Jul 08 2016 12:38:36 GMT+0000 (UTC)"
10   }
```

As you can see, the time difference is already present in the output from the getReplicationInfo function. You can choose to use either the timeDiff in seconds or **timeDiffHours** in hours here. A side note: this function is only available from the ***mongo*** command line tool.

It is easy to replicate this function by retrieving the first and last items from the oplog. Making use of the MongoDB aggregate function in a one single query is tempting, however the oplog does not have any indexes set on any of the fields. Running an aggregate function on a collection without indexes would require a full collection scan, which would become very slow in an oplog that has a couple of million entries.

Instead we are going to send two individual queries: fetch the first record of the oplog in forward and reverse order. As the oplog already is a sorted collection, we can naturally sort on the reverse of the collection cheaply.

```
1    mongo_replica_2:PRIMARY> use local
2    switched to db local
3    mongo_replica_2:PRIMARY> db.oplog.rs.find().limit(1);
4    { "ts" : Timestamp(1476312539, 1), "h" : Number-
     Long("-3302015507277893447"), "v" : 2, "op" : "n", "ns" :
     "", "o" : { "msg" : "initiating set" } }
5    mongo_replica_2:PRIMARY> db.oplog.rs.find().sort({$natural:
     -1}).limit(1);
6    { "ts" : Timestamp(1476403762, 1), "h" : Number-
     Long("3526317830277016106"), "v" : 2, "op" : "n",  "ns" :
     "ycsb.usertable", "o" : { "_id" : "user5864876345352853020",
7    …
8    }
```

The overhead of both queries is very low and will not interfere with the functioning of the oplog. In the example above, the replication window would be 91223 seconds (the difference of 1476403762 and 1476312539).

Intuitively you may think it only makes sense to do this calculation on the primary node, as this is the source for all write operations. However, MongoDB is a bit smarter than just serving out the oplog to **all** secondaries. Even though the secondary nodes will copy entries of the oplog from the primary, for joining members it will offload the delta of transactions loading via secondaries if possible. Also secondary nodes may prefer to fetch oplog entries from other secondaries with low latency, rather than fetching them from a primary with high latency. So it would be better to perform this calculation on **all** nodes in the cluster.

Another important metric is the replication lag. It suffices to connect to the primary and retrieve this data using the replSetGetStatus command, as the primary keeps track of the replication status of its secondaries.

A condensed version of this command is seen below:

```
my_mongodb_0:PRIMARY> db.runCommand( { replSetGetStatus: 1 } )
{
…
    “members” : [
        {
            “_id” : 0,
            “name” : “10.10.32.11:27017”,
            “stateStr” : “PRIMARY”,
            “optime” : {
                “ts” : Timestamp(1466247801, 5),
                “t” : NumberLong(1)
            },
            “optimeDate” : ISODate(“2016-06-18T11:03:21Z”),
        },
        {
            “_id” : 1,
            “name” : “10.10.32.12:27017”,
            “stateStr” : “SECONDARY”,
            “optime” : {
                “ts” : Timestamp(1466247801, 5),
                “t” : NumberLong(1)
            },
            “optimeDate” : ISODate(“2016-06-18T11:03:21Z”),
        },
        {
            “_id” : 2,
            “name” : “10.10.32.13:27017”,
            “stateStr” : “SECONDARY”,
            “optime” : {
                “ts” : Timestamp(1466247801, 5),
                “t” : NumberLong(1)
            },
            “optimeDate” : ISODate(“2016-06-18T11:03:21Z”),
        }
    ],
    “ok” : 1
}
```

You can calculate the lag by simply subtracting the secondary optimeDate (or optime timestamp) from the primary optimeDate. This will give you the replication lag in seconds.

4.5. MongoDB locks

MongoDB does support Global, Database and Collection level locking and will also report this in the serverStatus output:

```
mongo_replica_0:PRIMARY> db.serverStatus().locks
{
    "Global" : {
        "acquireCount" : {
            "r" : NumberLong(2667294),
            "w" : NumberLong(20),
            "R" : NumberLong(1),
            "W" : NumberLong(7)
        },
        "acquireWaitCount" : {
            "r" : NumberLong(1),
            "w" : NumberLong(1),
            "W" : NumberLong(1)
        },
        "timeAcquiringMicros" : {
            "r" : NumberLong(2101),
            "w" : NumberLong(4443),
            "W" : NumberLong(52)
        }
    },
    "Database" : {
        "acquireCount" : {
            "r" : NumberLong(1333616),
            "w" : NumberLong(8),
            "R" : NumberLong(17),
            "W" : NumberLong(12)
        }
    },
    "Collection" : {
        "acquireCount" : {
            "r" : NumberLong(678231),
            "w" : NumberLong(1)
        }
    },
    "Metadata" : {
        "acquireCount" : {
            "w" : NumberLong(7)
        }
    },
    "oplog" : {
        "acquireCount" : {
            "r" : NumberLong(678288),
            "w" : NumberLong(8)
        }
    }
}
```

In principle in MongoDB you should not see much locking happening as these locks are comparable global, schema and table type of locks. The document level locking is missing here as these locks are handled by the storage engine used. In the case of MMAPv1 (< MongoDB v3.0) the locks will happen on database level.

You should collect each and every one of these metrics as they might help you find performance issues outside the storage engines.

4.6. WiredTiger metrics

4.6.1. Locks and concurrency

As described in the previous section, the document level locking is handled by the storage engine. In the case of WiredTiger, it has locks to prevent one thread from writing to the same document as another thread. When a write occurs, a ticket is created to perform the write operation, where the ticket is comparable to a thread.

The number of concurrent transactions are reflected in the wiredTiger.concurrentTransactions metric:

```
1    mongo_replica_0:PRIMARY> db.serverStatus().wiredTiger.con-
     currentTransactions
2    {
3        "write" : {
4            "out" : 0,
5            "available" : 128,
6            "totalTickets" : 128
7        },
8        "read" : {
9            "out" : 0,
10           "available" : 128,
11           "totalTickets" : 128
12       }
13   }
```

This metric is important because of two reasons: if you see a sudden increase in the write.out tickets, there is probably a lot of document locking going on. Also if the **read.available** or **write.available** metrics are nearing zero your threads are getting exhausted. The result will be that new incoming requests are going to be queued.

4.6.2. Transactions

In contrary to the default MongoDB metrics, the WiredTiger output in serverStatus does contain information about transactions.

```
1    mongo_replica_0:PRIMARY> db.serverStatus().wiredTiger.trans-
     action
2    {
3        "number of named snapshots created" : 0,
4        "number of named snapshots dropped" : 0,
5        "transaction begins" : 21,
6        "transaction checkpoint currently running" : 0,
7        "transaction checkpoint generation" : 4610,
8        "transaction checkpoint max time (msecs)" : 12,
```

```
9           “transaction checkpoint min time (msecs)” : 0,
10          “transaction checkpoint most recent time (msecs)” : 0,
11          “transaction checkpoint total time (msecs)” : 6478,
12          “transaction checkpoints” : 4610,
13          “transaction failures due to cache overflow” : 0,
14          “transaction range of IDs currently pinned” : 1,
15          “transaction range of IDs currently pinned by a check-
        point” : 0,
16          “transaction range of IDs currently pinned by named
        snapshots” : 0,
17          “transaction sync calls” : 0,
18          “transactions committed” : 14,
19          “transactions rolled back” : 7
20      }
```

Metrics to keep an eye on are the trends in **begins, committed** and **rolled back.**

At the same time you can extract the **checkpoint max time, checkpoint min time** and **checkpoint most recent time** here. If the checkpointing time taken starts to increase WiredTiger isn't able to checkpoint the data as quickly as before. It would be best to correlate this with disk statistics.

Backup and Recovery

Backups in MongoDB aren't that different from MySQL backups.You have to start a copy process, ship the files to a safe place and ensure the backup is consistent. The consistency is obviously the biggest concern, as MongoDB doesn't feature a transaction mode that allows you to create a consistent snapshot. Obviously there are other ways to ensure we make a consistent backup.

There are two categories of backups available for MongoDB: logical and physical backups. The logical backups are basically data dumps from MongoDB, while the physical backups are copies of the data on disk.

5.1. Logical backups

All logical backup methods will not make a consistent backup, not without putting a global lock on the node you're making a backup of. This is comparable to mysqldump with MyISAM tables. This means it would be best to make a logical backup from a secondary node and set a global lock to ensure consistency.

For MongoDB there is a mysqldump equivalent: **mongodump**. This command line tool is shipped with every MongoDB installation and allows you to dump the contents of your MongoDB node into a BSON formatted dump file. BSON is a binary variant of JSON and this will not only keep the dump compact, but also improves recovery time.

5.2. Physical backups

For physical backups, there is no out of the box solution. Options here are to use the existing LVM, ZFS and EBS snapshot solutions. For LVM and ZFS, the snapshotting will freeze the file system in operation. However for EBS, a consistent snapshot can't be created unless writes have been stopped.

To do so, you have to fsync everything to disk and set a global lock:

```
1   my_mongodb_0:PRIMARY> use admin
2   switched to db admin
3   my_mongodb_0:PRIMARY> db.runCommand({fsync:1,lock:1});
4   {
5       "info" : "now locked against writes, use db.fsyncUn-
    lock() to unlock",
6       "seeAlso" : "http://dochub.mongodb.org/core/fsynccom-
    mand",
7       "ok" : 1
8   }
```

Don't forget to unlock after completing the EBS snapshot:

```
1    my_mongodb_0:PRIMARY> db.fsyncUnlock()
2    { "info" : "unlock completed", "ok" : 1 }
```

As MongoDB only checkpoints every 60 seconds, this means you will have to also include the journals. If these journals are not on the same disk, your snapshot may not be 100% consistent. This would be similar as making an LVM snapshot of a disk only containing the MySQL data without the redo logs.

If you are using MongoRocks, you also have the possibility to make a physical copy of all the data using the Strata backup tool. The Strata command line tool allows you to create a full backup or incremental backup. The best part of the Strata backup is that these physical files are queryable via mongo shell. This means you can utilize physical copies of your data to load data into your data warehouse or big data systems.

5.3. Sharded MongoDB backups

As the sharded MongoDB cluster consists of multiple replicaSets, a config replicaSet and Shard servers, it is very difficult to make a consistent backup. As every replicaSet is decoupled from each other, it is almost impossible to snapshot everything at the same time. Ideally a sharded MongoDB cluster should be frozen for a brief moment in time, and then a consistent backup taken. However this strategy would cause global locks and this means your clients will experience downtime.

The Percona Consistent Backup tool will take care of this problem: it will start all backups simultaneously and keeps copying the oplog entries throughout the whole process until the last node in the cluster is done.

5.4. Backup strategies

Ensure backups are being made, so check your backup on a regular interval (daily, weekly). Make sure the size of the backups makes sense and the logs are clear from errors. You could also check the integrity of the backup by extracting it and making a couple of checks on data points or files that need to be present. Automation for this process makes your life easier.

5.4.1. Offsite backups

There are many reasons for shipping your backups to another location. The best known reason may be (disaster) recovery, but other good reasons are keeping local copies for testing or data loading to offload the production database.

You could send your backups, for instance, to another datacenter or Amazon S3 or Glacier. To automatically ship your backups to a second location, you could use BitTorrent Sync. If you ship your backups to a less trusted location, you must store your backups encrypted.

5.4.2. Backup encryption

Even if you are keeping your backups in your local datacenter, it is still a good practice to encrypt them. Encrypting the backups will ensure nobody, unless they have the key, will be able to read them. Especially backups made using Strata will be partly readable, without the necessity to start up MongoDB. But also dumps via Mongodump and filesystem snapshots will be partly readable. So consider MongoDB backups to be insecure and always encrypt them. Storing them in a cloud even makes the necessity for encryption bigger.

5.4.3. Recovery

In addition to the health checks, also try to restore a backup on a regular (monthly) basis to verify if you can recover from a backup. This process includes extracting/decrypting the backup, starting up a new instance and possibly starting replication from the primary. This will give you a good indication whether your backups are in good condition. If you don't have a disaster recovery plan yet, make one and make sure these procedures are part of it.

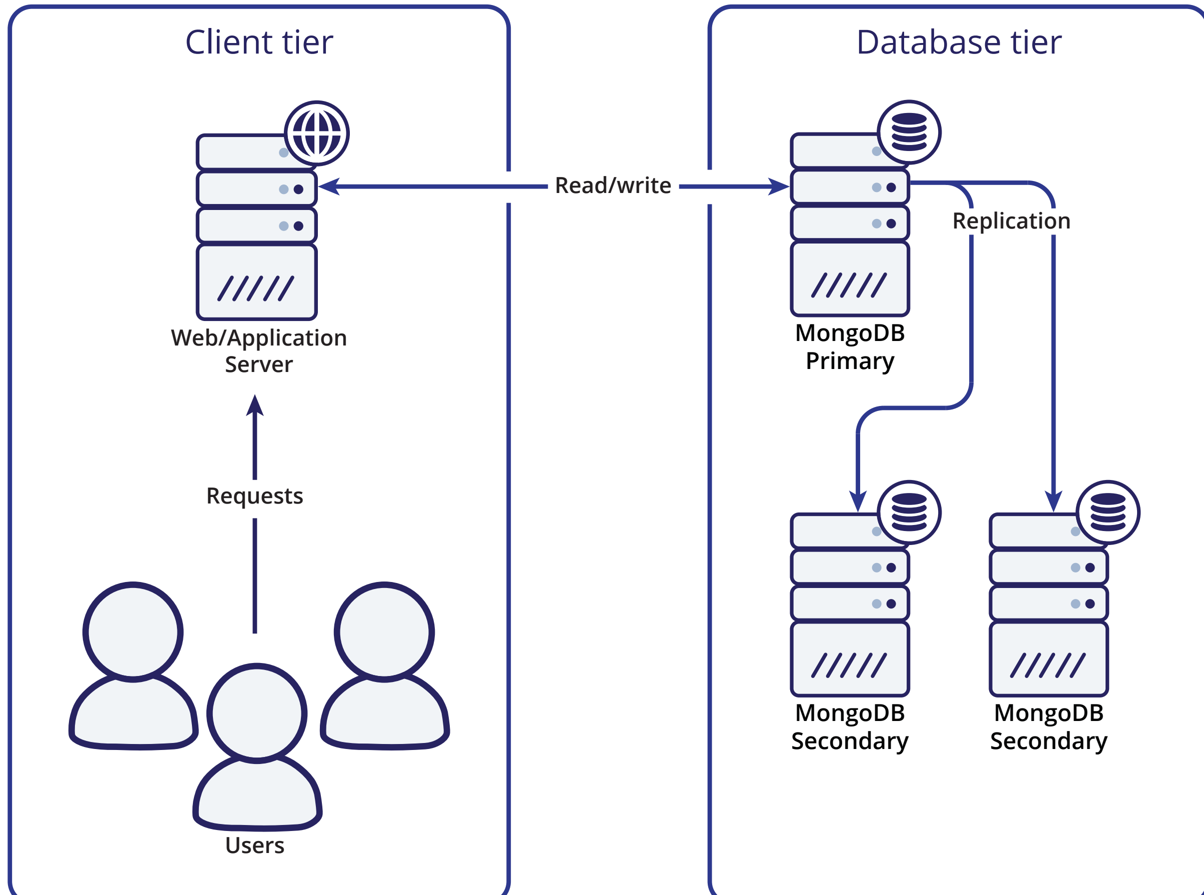
Scaling MongoDB

One of the cornerstones of MongoDB is that it is built with high availability and scaling in mind. Scaling can be done either vertically (bigger hardware) or horizontally (more nodes). Horizontal scaling is what MongoDB is good at, and it is not much more than spreading the workload to multiple machines. In effect, we're making use of multiple low-cost commodity hardware boxes, rather than upgrading to a more expensive high performance server.

MongoDB offers both read- and write scaling, and we will uncover the differences of these two strategies for you. Whether to choose read- or write scaling all depends on the workload of your application: if your application tends to read more often than it writes data you will probably want to make use of the read scaling capabilities of MongoDB.

6.1. Read scaling

With read scaling, we will scale out our read capacity. If you have used MongoDB before, you may be aware that actually all reads end up on the primary by default. Regardless if your replicaSet contains nine nodes, your read requests still go to the primary. Why was this done deliberately?

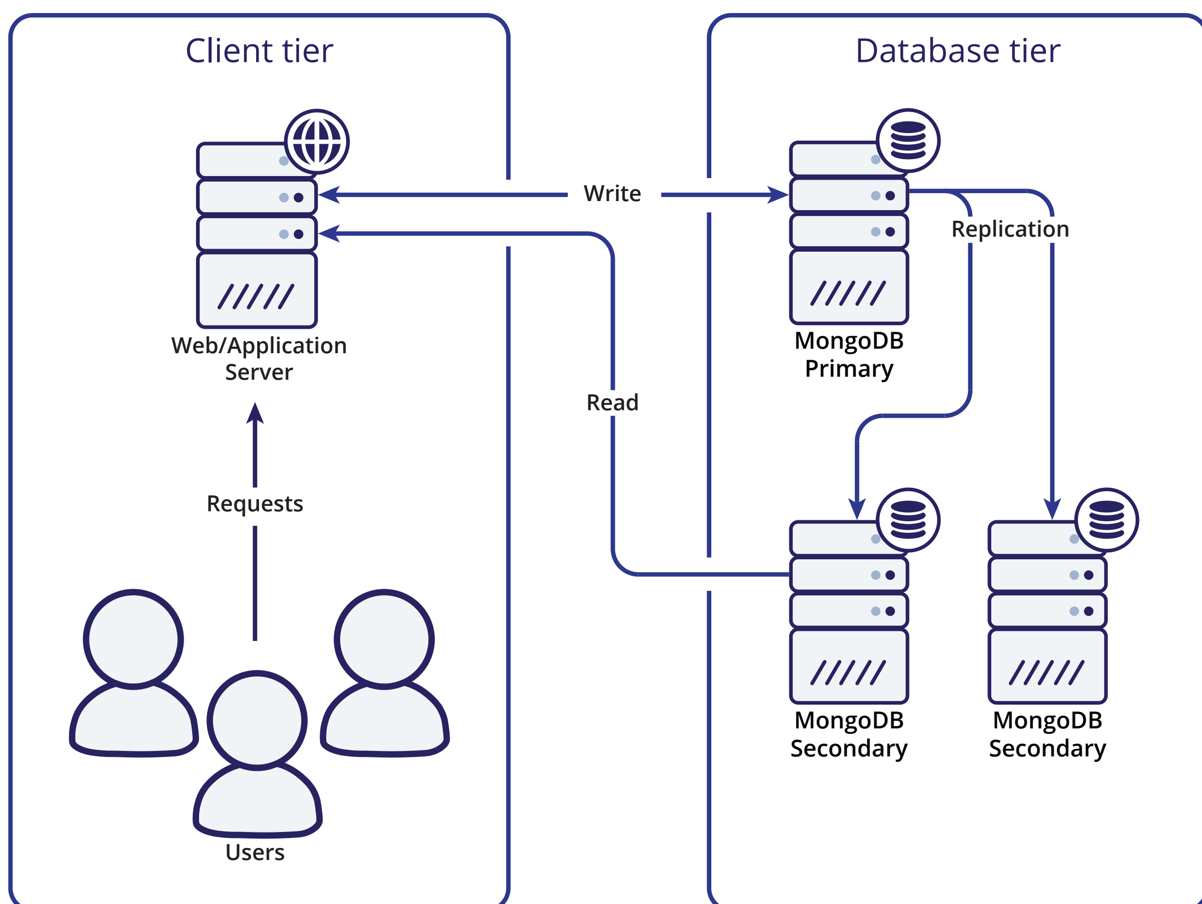


In principle, there are a few considerations to make before you start reading from a secondary node directly. First of all: the replication is asynchronous, so not all secondaries will give the same results if you read the same data at the same point in time. Secondly: if you distribute read requests to all secondaries and use up too much of their capacity, if one of them becomes unavailable, the other secondaries may not be able to cope with the extra workload. Thirdly: on sharded clusters you should never bypass the shard router, as data may be out-of-date or data may have been moved to another shard. If you do use the shard router and set the read preference correctly, it may still return incorrect data due to incomplete or terminated chunk migrations.

As you have seen these are serious considerations you should make before scaling out your read queries on MongoDB. In general, unless your primary is not able to cope with the read workload it is receiving, we would advise against reading from secondaries. The price you pay for inconsistency is relatively high, compared to the benefits of offloading work from the master.

6.2. Reading from a secondary

There are two things that are necessary to make reading from a secondary possible: tell the MongoDB client driver that you actually wish to read from a secondary (if possible) and tell the MongoDB secondary server that it is okay to read from this node.

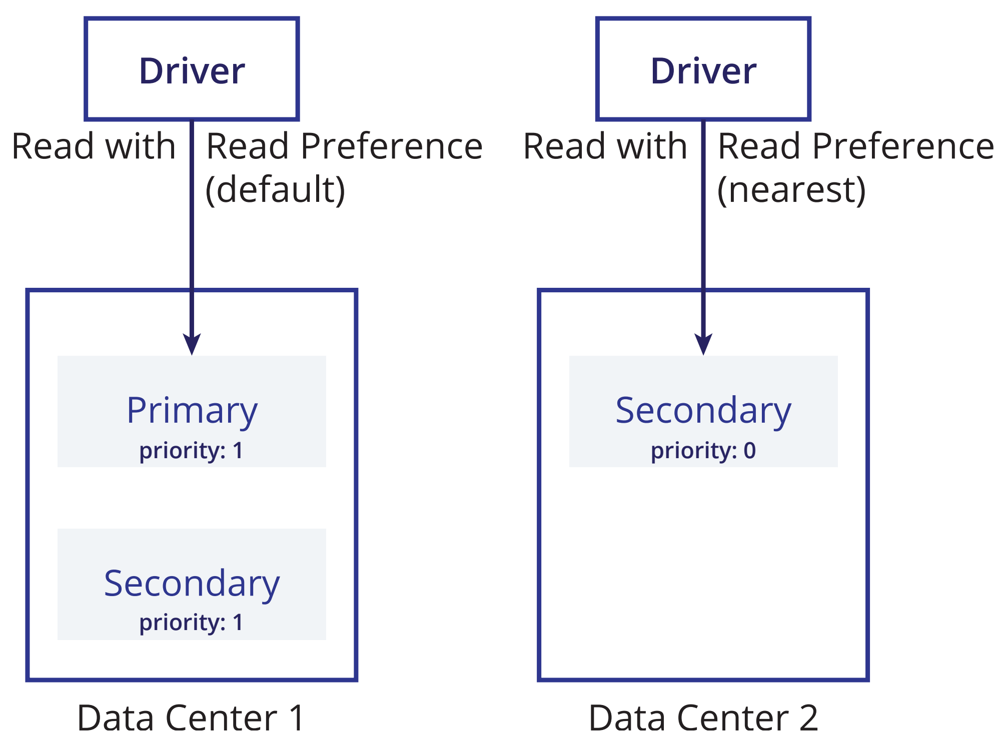


6.2.1. Setting read preference

For the driver, all you have to do is set the read preference. When reading data you simply set the read preference to read from a secondary. Let's go over each and every read preference and explain what it does:

primary	Always read from the primary (default)
primaryPreferred	Always read from the primary, read from secondary if the primary is unavailable
secondary	Always read from a secondary
secondaryPreferred	Always read from a secondary, read from the primary if no secondary is available
nearest	Always read from the node with the lowest network latency

It is clear the default mode is the least preferred if you wish to scale out reads. PrimaryPreferred is not much better, as in 99.999% of the time, it will pick the primary. Still if the primary becomes unavailable you will have a fallback for read requests.



Secondary should work fine for scaling reads, but as you leave out the primary the reads will never have a fallback if no secondary is available. SecondaryPreferred is slightly better, but the reads will hit almost all of the time the secondaries, which still causes an uneven spread of reads. Also if no secondaries are available, in most cases there will be no longer a cluster and the primary will demote itself to a secondary. Only when an arbiter is part of the cluster, the secondaryPreferred mode makes sense.

Nearest should always pick the node with the lowest network latency. Even though this sounds great from an application perspective, this will not guarantee you get an even spread in read operations. But it will work very well in multi-regions where latency is high, and delays are noticeable. In such cases, reading from the nearest node means your application will be able to serve out data with the minimum latency.

6.2.2. Reading from a secondary in a shard

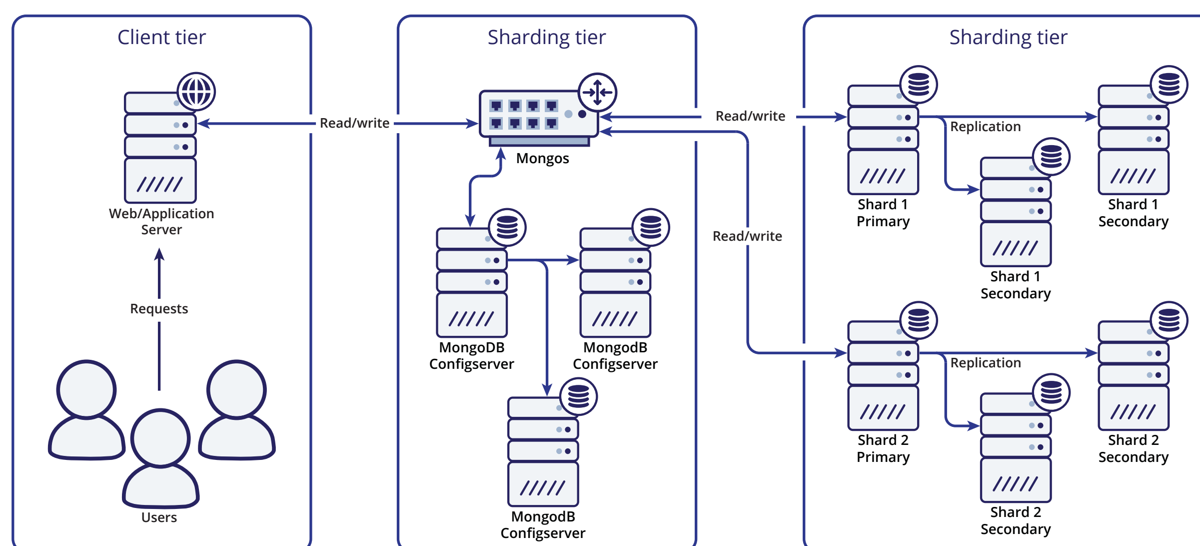
It is also possible to read from a secondary node in MongoDB sharded clusters. The MongoDB shard router (mongos) will obey the read preference set in the request and forward the request to a secondary in the shard(s). This also means you will have to enable reads from a secondary on all hosts in the sharded environment.

And as said earlier: an issue that may arise with reading from secondaries on a sharded environment, is that it might be possible to receive incorrect data from a secondary. Due to the migration of data between shards, data may be in transit from one shard to another. Reading from a secondary may then return incomplete data, therefore we would strongly recommend against performing secondary reads in a sharded cluster.

6.3. MongoDB write scaling (sharding)

The MongoDB sharding solution is similar to existing sharding frameworks for other major database solutions. It makes use of a typical lookup solution, where the sharding is defined in a shard-key and the ranges are stored inside a configuration database. MongoDB works with three components to find the correct shard for your data.

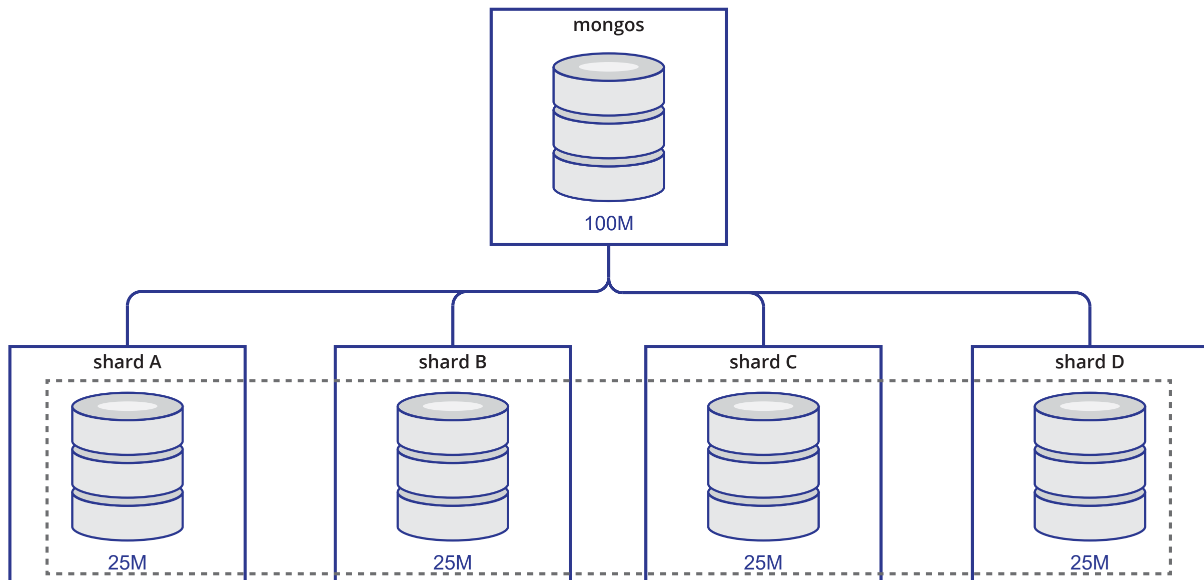
A typical sharded MongoDB environment looks like this:



6.3.1. Sharding tier

The first component used is the shard router called **mongos**. All read and write operations must be sent to the shard router, making all shards act as a single database for the client application. The shard router will route the queries to the appropriate shards by consulting the **Configserver**.

The Configserver is a special replicaSet that keeps the configuration of all shards in the cluster. The Configserver contains information about shards, databases, collections, shard keys and the distribution of chunks of data. Data gets partitioned by slicing the total dataset into smaller chunks of data, where these chunks are defined by the shard key. The shard key can be either range or hash defined. These chunks are then distributed evenly over the total number of shards.



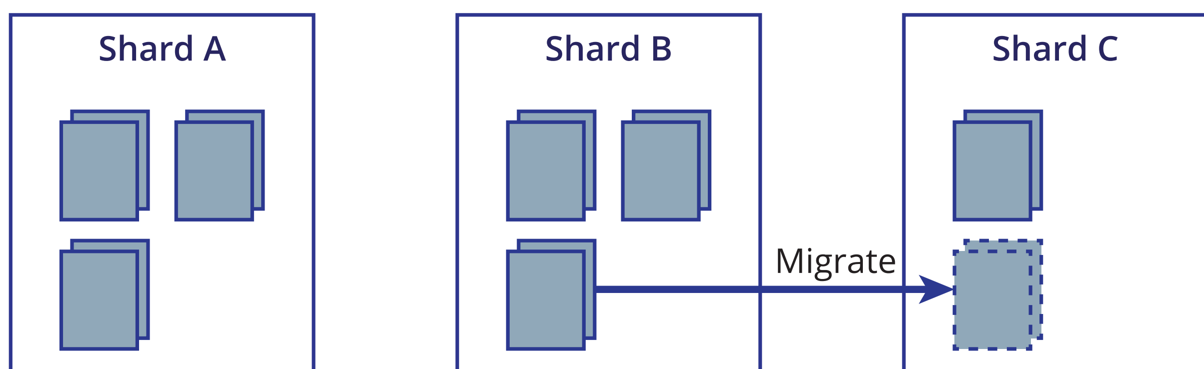
The router will know on which shard to place the data by finding the correct chunk in the Configserver. If the router thinks the chunk is becoming too large, it will automatically create a new chunk in the Configserver. The sharding metadata is stored in the **config** database, and this database is accessible via the shard router as well.

Prior to MongoDB 3.2 the Configserver used to be a total of three individual MongoDB nodes that were used to write the sharding metadata. In this setup the metadata is written and read thrice, and differences in data between nodes means inconsistent writes happened and will require manual intervention. If this happened, the balancer would no longer perform shard migrations and the shard router was no longer able to create new chunks.

6.3.2. Shard tier

Each replicaSet in a MongoDB sharded cluster is treated as an individual **shard**. Adding a shard will increase the write capacity, but also increase the sharding complexity. Each shard is an individual component in the cluster and there is no direct communication between them. Shards don't know anything about other shards in the cluster.

MongoDB distributes its data evenly by balancing the total number of chunks on each shard. If the number of chunks is not spread evenly, a balancing process can be run to migrate chunks from one shard to another.

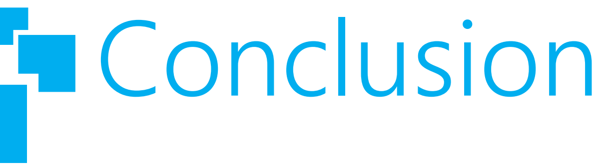


On MongoDB 3.2 and older this balancing process typically gets started from a shard router (**mongos**), that thinks the data is unbalanced. The shard router will acquire and set a lock on the balancing process in the config database on the Configserver. This lock is necessary as there may be multiple shard routers active inside the same cluster.

To overcome this problem, starting from MongoDB 3.4 onwards, this process will be started on the primary of the Configserver.

6.3.3. Managing shards

Shard management is really easy in MongoDB. You can add and remove shards online and the MongoDB shard router will automatically adjust to what you tell it to. If you wish to know more in depth about how best to manage shards, please read our blog post about managing MongoDB shards.



Conclusion

In this whitepaper we have collected some of the best practices for running MongoDB in production. More tips and tricks are available from our Become a MongoDB DBA blog series and webinars. We would recommend reading the full blog series to learn more about deploying, monitoring, managing and scaling MongoDB.

About ClusterControl

ClusterControl is the all-inclusive open source database management system for users with mixed environments that removes the need for multiple management tools. ClusterControl provides advanced deployment, management, monitoring, and scaling functionality to get your MySQL, MongoDB, and PostgreSQL databases up-and-running using proven methodologies that you can depend on to work. At the core of ClusterControl is it's automation functionality that let's you automate many of the database tasks you have to perform regularly like deploying new databases, adding and scaling new nodes, running backups and upgrades, and more.

About Severalnines

Severalnines provides automation and management software for database clusters. We help companies deploy their databases in any environment, and manage all operational aspects to achieve high-scale availability.

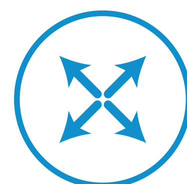
Severalnines' products are used by developers and administrators of all skills levels to provide the full 'deploy, manage, monitor, scale' database cycle, thus freeing them from the complexity and learning curves that are typically associated with highly available database clusters. Severalnines is often called the "anti-startup" as it is entirely self-funded by its founders. The company has enabled over 12,000 deployments to date via its popular product ClusterControl. Currently counting BT, Orange, Cisco, CNRS, Technicolor, AVG, Ping Identity and Paytrail as customers. Severalnines is a private company headquartered in Stockholm, Sweden with offices in Singapore, Japan and the United States. To see who is using Severalnines today visit:

https://www.severalnines.com/company

Deploy

Manage

Monitor



Scale

Related Resources from Severalnines

Watch the MongoDB Webinars

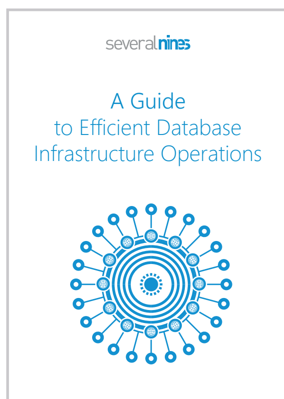
We have produced many hours of MongoDB webinar content, some of which are in-depth topics that were discussed in this whitepaper. All of our webinar replays are online and available for viewing for free!

Watch Now

Become a MongoDB DBA Blog Series

Read the blog series that inspired the whitepaper! The Become a MongoDB DBA blog series covers all you need to help you deploy, monitor, manage and scale MongoDB in your environment.

Read the Blogs



A Guide to Efficient Database Infrastructure Operations

Taking control of their data is every company's number one job. Database operations encompass a number of functions, including the initial deployment of a solution, configuration management, performance monitoring, SLA management, backups, patches, version upgrades and scaling.

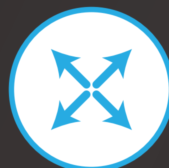
Download whitepaper

severalnines

Deploy

Manage

Monitor



Scale

© 2017 Severalnines AB. All rights reserved. Severalnines and the Severalnines logo(s) are
trademarks of Severalnines AB. Other names may be trademarks of their respective owners.