MongoDB Administration

Release 3.0.8

MongoDB, Inc.

January 16, 2016

Contents

The administration documentation addresses the ongoing operation and maintenance of MongoDB instances and deployments. This documentation includes both high level overviews of these concerns as well as tutorials that cover specific procedures and processes for operating MongoDB.

class `hidden`

CHAPTER 1

Administration Concepts

The core administration documents address strategies and practices used in the operation of MongoDB systems and deployments.

***Operational Strategies* (page 3)** Higher level documentation of key concepts for the operation and maintenance of MongoDB deployments.

***MongoDB Backup Methods* (page 4)** Describes approaches and considerations for backing up a MongoDB database.

***Monitoring for MongoDB* (page 6)** An overview of monitoring tools, diagnostic strategies, and approaches to monitoring replica sets and sharded clusters.

***Run-time Database Configuration* (page 12)** Outlines common MongoDB configurations and examples of best-practice configurations for common use cases.

Continue reading from *Operational Strategies* (page 3) for additional documentation.

***Data Management* (page 29)** Core documentation that addresses issues in data management, organization, maintenance, and lifecycle management.

***Data Center Awareness* (page 30)** Presents the MongoDB features that allow application developers and database administrators to configure their deployments to be more data center aware or allow operational and location-based separation.

***Capped Collections* (page 32)** Capped collections provide a special type of size-constrained collections that preserve insertion order and can support high volume inserts.

***Expire Data from Collections by Setting TTL* (page 35)** TTL collections make it possible to automatically remove data from a collection based on the value of a timestamp and are useful for managing data like machine generated event data that are only useful for a limited period of time.

***Optimization Strategies for MongoDB* (page 37)** Techniques for optimizing application performance with MongoDB.

Continue reading from *Optimization Strategies for MongoDB* (page 37) for additional documentation.

1.1 Operational Strategies

These documents address higher level strategies for common administrative tasks and requirements with respect to MongoDB deployments.

***MongoDB Backup Methods* (page 4)** Describes approaches and considerations for backing up a MongoDB database.

Monitoring for MongoDB **(page 6)** An overview of monitoring tools, diagnostic strategies, and approaches to monitoring replica sets and sharded clusters.

Run-time Database Configuration **(page 12)** Outlines common MongoDB configurations and examples of best-practice configurations for common use cases.

Production Notes **(page 18)** A collection of notes that describe best practices and considerations for the operations of MongoDB instances and deployments.

1.1.1 MongoDB Backup Methods

On this page

When deploying MongoDB in production, you should have a strategy for capturing and restoring backups in the case of data loss events. There are several ways to back up MongoDB clusters:

- *Backup by Copying Underlying Data Files* (page 4)
- *Backup a Database with mongodump* (page 83)
- *MongoDB Cloud Manager Backup* (page 5)
- *Ops Manager Backup Software* (page 6)

Backup by Copying Underlying Data Files

You can create a backup by copying MongoDB's underlying data files.

If the volume where MongoDB stores data files supports point in time snapshots, you can use these snapshots to create backups of a MongoDB system at an exact moment in time.

File systems snapshots are an operating system volume manager feature, and are not specific to MongoDB. The mechanics of snapshots depend on the underlying storage system. For example, if you use Amazon's EBS storage system for EC2 supports snapshots. On Linux the LVM manager can create a snapshot.

To get a correct snapshot of a running `mongod` process, you must have journaling enabled and the journal must reside on the same logical volume as the other MongoDB data files. Without journaling enabled, there is no guarantee that the snapshot will be consistent or valid.

To get a consistent snapshot of a sharded system, you must disable the balancer and capture a snapshot from every shard and a config server at approximately the same moment in time.

If your storage system does not support snapshots, you can copy the files directly using `cp`, `rsync`, or a similar tool. Since copying multiple files is not an atomic operation, you must stop all writes to the `mongod` before copying the files. Otherwise, you will copy the files in an invalid state.

Backups produced by copying the underlying data do not support point in time recovery for replica sets and are difficult to manage for larger sharded clusters. Additionally, these backups are larger because they include the indexes and duplicate underlying storage padding and fragmentation. `mongodump`, by contrast, creates smaller backups.

For more information, see the *Backup and Restore with Filesystem Snapshots* (page 75) and *Backup a Sharded Cluster with Filesystem Snapshots* (page 90) for complete instructions on using LVM to create snapshots. Also see Back up and Restore Processes for MongoDB on Amazon EC2[1].

Backup with `mongodump`

The `mongodump` tool reads data from a MongoDB database and creates high fidelity BSON files. The `mongorestore` tool can populate a MongoDB database with the data from these BSON files. These tools are simple and efficient for backing up small MongoDB deployments, but are not ideal for capturing backups of larger systems.

`mongodump` and `mongorestore` operate against a running `mongod` process, and can manipulate the underlying data files directly. By default, `mongodump` does not capture the contents of the `local database`.

`mongodump` only captures the documents in the database. The resulting backup is space efficient, but `mongorestore` or `mongod` must rebuild the indexes after restoring data.

When connected to a MongoDB instance, `mongodump` can adversely affect `mongod` performance. If your data is larger than system memory, the queries will push the working set out of memory.

To mitigate the impact of `mongodump` on the performance of a replica set, use `mongodump` to capture backups from a `secondary` member of the replica set.

For replica sets, `mongodump` also supports a point in time feature with the `--oplog` option. Applications may continue modifying data while `mongodump` captures the output. To restore a point in time backup created with `--oplog`, use `mongorestore` with the `--oplogReplay` option.

If applications modify data while `mongodump` is creating a backup, `mongodump` will compete for resources with those applications.

See *Back Up and Restore with MongoDB Tools* (page 82), *Backup a Small Sharded Cluster with mongodump* (page 88), and *Backup a Sharded Cluster with Database Dumps* (page 92) for more information.

MongoDB Cloud Manager Backup

The MongoDB Cloud Manager[2] supports the backing up and restoring of MongoDB deployments.

MongoDB Cloud Manager continually backs up MongoDB replica sets and sharded clusters by reading the oplog data from your MongoDB deployment.

MongoDB Cloud Manager Backup offers point in time recovery of MongoDB replica sets and a consistent snapshot of sharded clusters.

MongoDB Cloud Manager achieves point in time recovery by storing oplog data so that it can create a restore for any moment in time in the last 24 hours for a particular replica set or sharded cluster. Sharded cluster snapshots are difficult to achieve with other MongoDB backup methods.

To restore a MongoDB deployment from an MongoDB Cloud Manager Backup snapshot, you download a compressed archive of your MongoDB data files and distribute those files before restarting the `mongod` processes.

To get started with MongoDB Cloud Manager Backup, sign up for MongoDB Cloud Manager[3]. For documentation on MongoDB Cloud Manager, see the MongoDB Cloud Manager documentation[4].

[1] https://docs.mongodb.org/ecosystem/tutorial/backup-and-restore-mongodb-on-amazon-ec2
[2] https://cloud.mongodb.com/?jmp=docs
[3] https://cloud.mongodb.com/?jmp=docs
[4] https://docs.cloud.mongodb.com/

Ops Manager Backup Software

MongoDB Subscribers can install and run the same core software that powers *MongoDB Cloud Manager Backup* (page 5) on their own infrastructure. Ops Manager, an on-premise solution, has similar functionality to the cloud version and is available with Enterprise Advanced subscriptions.

For more information about Ops Manager, see the MongoDB Enterprise Advanced[5] page and the Ops Manager Manual[6].

Further Reading

Backup and Restore with Filesystem Snapshots **(page 75)** An outline of procedures for creating MongoDB data set backups using system-level file snapshot tool, such as *LVM* or native storage appliance tools.

Restore a Replica Set from MongoDB Backups **(page 80)** Describes procedure for restoring a replica set from an archived backup such as a `mongodump` or MongoDB Cloud Manager[7] Backup file.

Back Up and Restore with MongoDB Tools **(page 82)** Describes a procedure for exporting the contents of a database to either a binary dump or a textual exchange format, and for importing these files into a database.

Backup and Restore Sharded Clusters **(page 88)** Detailed procedures and considerations for backing up sharded clusters and single shards.

Recover Data after an Unexpected Shutdown **(page 98)** Recover data from MongoDB data files that were not properly closed or have an invalid state.

Additional Resources

- Backup and it's Role in Disaster Recovery White Paper[8]
- Backup vs. Replication: Why Do You Need Both?[9]
- MongoDB Production Readiness Consulting Package[10]

1.1.2 Monitoring for MongoDB

On this page

[5] https://www.mongodb.com/products/mongodb-enterprise-advanced?jmp=docs
[6] https://docs.opsmanager.mongodb.com/current/
[7] https://cloud.mongodb.com/?jmp=docs
[8] https://www.mongodb.com/lp/white-paper/backup-disaster-recovery?jmp=docs
[9] http://www.mongodb.com/blog/post/backup-vs-replication-why-do-you-need-both?jmp=docs
[10] https://www.mongodb.com/products/consulting?jmp=docs#s_product_readiness

Monitoring is a critical component of all database administration. A firm grasp of MongoDB's reporting will allow you to assess the state of your database and maintain your deployment without crisis. Additionally, a sense of MongoDB's normal operational parameters will allow you to diagnose problems before they escalate to failures.

This document presents an overview of the available monitoring utilities and the reporting statistics available in MongoDB. It also introduces diagnostic strategies and suggestions for monitoring replica sets and sharded clusters.

Note: MongoDB Cloud Manager[11], a hosted service, and Ops Manager[12], an on-premise solution, provide monitoring, backup, and automation of MongoDB instances. See the MongoDB Cloud Manager documentation[13] and Ops Manager documentation[14] for more information.

Monitoring Strategies

There are three methods for collecting data about the state of a running MongoDB instance:

- First, there is a set of utilities distributed with MongoDB that provides real-time reporting of database activities.
- Second, `database commands` return statistics regarding the current database state with greater fidelity.
- Third, MongoDB Cloud Manager[15], a hosted service, and Ops Manager, an on-premise solution available in MongoDB Enterprise Advanced[16], provide monitoring to collect data from running MongoDB deployments as well as providing visualization and alerts based on that data.

Each strategy can help answer different questions and is useful in different contexts. These methods are complementary.

MongoDB Reporting Tools

This section provides an overview of the reporting methods distributed with MongoDB. It also offers examples of the kinds of questions that each method is best suited to help you address.

Utilities

The MongoDB distribution includes a number of utilities that quickly return statistics about instances' performance and activity. Typically, these are most useful for diagnosing issues and assessing normal operation.

`mongostat` `mongostat` captures and returns the counts of database operations by type (e.g. insert, query, update, delete, etc.). These counts report on the load distribution on the server.

Use `mongostat` to understand the distribution of operation types and to inform capacity planning. See the `mongostat manual` for details.

[11] https://cloud.mongodb.com/?jmp=docs
[12] https://www.mongodb.com/products/mongodb-enterprise-advanced?jmp=docs
[13] https://docs.cloud.mongodb.com/
[14] https://docs.opsmanager.mongodb.com?jmp=docs
[15] https://cloud.mongodb.com/?jmp=docs
[16] https://www.mongodb.com/products/mongodb-enterprise-advanced?jmp=docs

mongotop `mongotop` tracks and reports the current read and write activity of a MongoDB instance, and reports these statistics on a per collection basis.

Use `mongotop` to check if your database activity and use match your expectations. See the `mongotop manual` for details.

HTTP Console MongoDB provides a web interface that exposes diagnostic and monitoring information in a simple web page. The web interface is accessible at `localhost:<port>`, where the `<port>` number is **1000** more than the `mongod` port .

For example, if a locally running `mongod` is using the default port `27017`, access the HTTP console at `http://localhost:28017`.

Commands

MongoDB includes a number of commands that report on the state of the database.

These data may provide a finer level of granularity than the utilities discussed above. Consider using their output in scripts and programs to develop custom alerts, or to modify the behavior of your application in response to the activity of your instance. The `db.currentOp` method is another useful tool for identifying the database instance's in-progress operations.

serverStatus The `serverStatus` command, or `db.serverStatus()` from the shell, returns a general overview of the status of the database, detailing disk usage, memory use, connection, journaling, and index access. The command returns quickly and does not impact MongoDB performance.

`serverStatus` outputs an account of the state of a MongoDB instance. This command is rarely run directly. In most cases, the data is more meaningful when aggregated, as one would see with monitoring tools including MongoDB Cloud Manager[17] and Ops Manager[18]. Nevertheless, all administrators should be familiar with the data provided by `serverStatus`.

dbStats The `dbStats` command, or `db.stats()` from the shell, returns a document that addresses storage use and data volumes. The `dbStats` reflect the amount of storage used, the quantity of data contained in the database, and object, collection, and index counters.

Use this data to monitor the state and storage capacity of a specific database. This output also allows you to compare use between databases and to determine the average *document* size in a database.

collStats The `collStats` or `db.collection.stats()` from the shell that provides statistics that resemble `dbStats` on the collection level, including a count of the objects in the collection, the size of the collection, the amount of disk space used by the collection, and information about its indexes.

replSetGetStatus The `replSetGetStatus` command (`rs.status()` from the shell) returns an overview of your replica set's status. The `replSetGetStatus` document details the state and configuration of the replica set and statistics about its members.

Use this data to ensure that replication is properly configured, and to check the connections between the current host and the other members of the replica set.

[17] https://cloud.mongodb.com/?jmp=docs
[18] https://www.mongodb.com/products/mongodb-enterprise-advanced?jmp=docs

Third Party Tools

A number of third party monitoring tools have support for MongoDB, either directly, or through their own plugins.

Self Hosted Monitoring Tools These are monitoring tools that you must install, configure and maintain on your own servers. Most are open source.

Tool	Plugin	Description
Ganglia[19]	mongodb-ganglia[20]	Python script to report operations per second, memory usage, btree statistics, master/slave status and current connections.
Ganglia	gmond_python_modules[21]	Parses output from the `serverStatus` and `replSetGetStatus` commands.
Motop[22]	*None*	Realtime monitoring tool for MongoDB servers. Shows current operations ordered by durations every second.
mtop[23]	*None*	A top like tool.
Munin[24]	mongo-munin[25]	Retrieves server statistics.
Munin	mongomon[26]	Retrieves collection statistics (sizes, index sizes, and each (configured) collection count for one DB).
Munin	munin-plugins Ubuntu PPA[27]	Some additional munin plugins not in the main distribution.
Nagios[28]	nagios-plugin-mongodb[29]	A simple Nagios check script, written in Python.

Also consider dex[30], an index and query analyzing tool for MongoDB that compares MongoDB log files and indexes to make indexing recommendations.

See also:

Ops Manager, an on-premise solution available in MongoDB Enterprise Advanced[31].

Hosted (SaaS) Monitoring Tools These are monitoring tools provided as a hosted service, usually through a paid subscription.

[19] http://sourceforge.net/apps/trac/ganglia/wiki
[20] https://github.com/quiiver/mongodb-ganglia
[21] https://github.com/ganglia/gmond_python_modules
[22] https://github.com/tart/motop
[23] https://github.com/beaufour/mtop
[24] http://munin-monitoring.org/
[25] https://github.com/erh/mongo-munin
[26] https://github.com/pcdummy/mongomon
[27] https://launchpad.net/ chris-lea/+archive/munin-plugins
[28] http://www.nagios.org/
[29] https://github.com/mzupan/nagios-plugin-mongodb
[30] https://github.com/mongolab/dex
[31] https://www.mongodb.com/products/mongodb-enterprise-advanced?jmp=docs

Name	Notes
MongoDB Cloud Manager[32]	MongoDB Cloud Manager is a cloud-based suite of services for managing MongoDB deployments. MongoDB Cloud Manager provides monitoring, backup, and automation functionality. For an on-premise solution, see also Ops Manager, available in MongoDB Enterprise Advanced[33].
Scout[34]	Several plugins, including MongoDB Monitoring[35], MongoDB Slow Queries[36], and MongoDB Replica Set Monitoring[37].
Server Density[38]	Dashboard for MongoDB[39], MongoDB specific alerts, replication failover timeline and iPhone, iPad and Android mobile apps.
Application Performance Management[40]	IBM has an Application Performance Management SaaS offering that includes monitor for MongoDB and other applications and middleware.

Process Logging

During normal operation, `mongod` and `mongos` instances report a live account of all server activity and operations to either standard output or a log file. The following runtime settings control these options.

- `quiet`. Limits the amount of information written to the log or output.
- `verbosity`. Increases the amount of information written to the log or output. You can also modify the logging verbosity during runtime with the `logLevel` parameter or the `db.setLogLevel()` method in the shell.
- `path`. Enables logging to a file, rather than the standard output. You must specify the full path to the log file when adjusting this setting.
- `logAppend`. Adds information to a log file instead of overwriting the file.

Note: You can specify these configuration operations as the command line arguments to `mongod` or `mongos`

For example:

```
mongod -v --logpath /var/log/mongodb/server1.log --logappend
```

Starts a `mongod` instance in `verbose` mode, appending data to the log file at `/var/log/mongodb/server1.log/`.

The following *database commands* also affect logging:

- `getLog`. Displays recent messages from the `mongod` process log.
- `logRotate`. Rotates the log files for `mongod` processes only. See *Rotate Log Files* (page 61).

Diagnosing Performance Issues

As you develop and operate applications with MongoDB, you may want to analyze the performance of the database as the application. *Analyzing MongoDB Performance* (page 37) discusses some of the operational factors that can influence performance.

[32] https://cloud.mongodb.com/?jmp=docs
[33] https://www.mongodb.com/products/mongodb-enterprise-advanced?jmp=docs
[34] http://scoutapp.com
[35] https://scoutapp.com/plugin_urls/391-mongodb-monitoring
[36] http://scoutapp.com/plugin_urls/291-mongodb-slow-queries
[37] http://scoutapp.com/plugin_urls/2251-mongodb-replica-set-monitoring
[38] http://www.serverdensity.com
[39] http://www.serverdensity.com/mongodb-monitoring/
[40] http://ibmserviceengage.com

Replication and Monitoring

Beyond the basic monitoring requirements for any MongoDB instance, for replica sets, administrators must monitor *replication lag*. "Replication lag" refers to the amount of time that it takes to copy (i.e. replicate) a write operation on the *primary* to a *secondary*. Some small delay period may be acceptable, but two significant problems emerge as replication lag grows:

- First, operations that occurred during the period of lag are not replicated to one or more secondaries. If you're using replication to ensure data persistence, exceptionally long delays may impact the integrity of your data set.
- Second, if the replication lag exceeds the length of the operation log (*oplog*) then MongoDB will have to perform an initial sync on the secondary, copying all data from the *primary* and rebuilding all indexes. This is uncommon under normal circumstances, but if you configure the oplog to be smaller than the default, the issue can arise.

Note: The size of the oplog is only configurable during the first run using the `--oplogSize` argument to the `mongod` command, or preferably, the `oplogSizeMB` setting in the MongoDB configuration file. If you do not specify this on the command line before running with the `--replSet` option, `mongod` will create a default sized oplog.

By default, the oplog is 5 percent of total available disk space on 64-bit systems. For more information about changing the oplog size, see the *Change the Size of the Oplog* (page 187)

For causes of replication lag, see *Replication Lag* (page 208).

Replication issues are most often the result of network connectivity issues between members, or the result of a *primary* that does not have the resources to support application and replication traffic. To check the status of a replica, use the `replSetGetStatus` or the following helper in the shell:

```
rs.status()
```

The `replSetGetStatus` reference provides a more in-depth overview view of this output. In general, watch the value of `optimeDate`, and pay particular attention to the time difference between the *primary* and the *secondary* members.

Sharding and Monitoring

In most cases, the components of *sharded clusters* benefit from the same monitoring and analysis as all other MongoDB instances. In addition, clusters require further monitoring to ensure that data is effectively distributed among nodes and that sharding operations are functioning appropriately.

See also:

See the `https://docs.mongodb.org/manual/core/sharding` documentation for more information.

Config Servers

The *config database* maintains a map identifying which documents are on which shards. The cluster updates this map as *chunks* move between shards. When a configuration server becomes inaccessible, certain sharding operations become unavailable, such as moving chunks and starting `mongos` instances. However, clusters remain accessible from already-running `mongos` instances.

Because inaccessible configuration servers can seriously impact the availability of a sharded cluster, you should monitor your configuration servers to ensure that the cluster remains well balanced and that `mongos` instances can restart.

MongoDB Cloud Manager[41] and Ops Manager[42] monitor config servers and can create notifications if a config server becomes inaccessible. See the MongoDB Cloud Manager documentation[43] and Ops Manager documentation[44] for more information.

Balancing and Chunk Distribution

The most effective *sharded cluster* deployments evenly balance *chunks* among the shards. To facilitate this, MongoDB has a background *balancer* process that distributes data to ensure that chunks are always optimally distributed among the *shards*.

Issue the `db.printShardingStatus()` or `sh.status()` command to the `mongos` by way of the `mongo` shell. This returns an overview of the entire cluster including the database name, and a list of the chunks.

Stale Locks

In nearly every case, all locks used by the balancer are automatically released when they become stale. However, because any long lasting lock can block future balancing, it's important to ensure that all locks are legitimate. To check the lock status of the database, connect to a `mongos` instance using the `mongo` shell. Issue the following command sequence to switch to the `config` database and display all outstanding locks on the shard database:

```
use config
db.locks.find()
```

For active deployments, the above query can provide insights. The balancing process, which originates on a randomly selected `mongos`, takes a special "balancer" lock that prevents other balancing activity from transpiring. Use the following command, also to the `config` database, to check the status of the "balancer" lock.

```
db.locks.find( { _id : "balancer" } )
```

If this lock exists, make sure that the balancer process is actively using this lock.

Additional Resources

- MongoDB Production Readiness Consulting Package[45]

1.1.3 Run-time Database Configuration

[41] https://cloud.mongodb.com/?jmp=docs
[42] https://www.mongodb.com/products/mongodb-enterprise-advanced?jmp=docs
[43] https://docs.cloud.mongodb.com/
[44] https://docs.opsmanager.mongodb.com/current/application
[45] https://www.mongodb.com/products/consulting?jmp=docs#s_product_readiness

On this page

The `command line` and `configuration file` interfaces provide MongoDB administrators with a large number of options and settings for controlling the operation of the database system. This document provides an overview of common configurations and examples of best-practice configurations for common use cases.

While both interfaces provide access to the same collection of options and settings, this document primarily uses the configuration file interface. If you run MongoDB using a *init script* or if you installed from a package for your operating system, you likely already have a configuration file located at `/etc/mongod.conf`. Confirm this by checking the contents of the `/etc/init.d/mongod` or `/etc/rc.d/mongod` script to ensure that the init scripts start the `mongod` with the appropriate configuration file.

To start a MongoDB instance using this configuration file, issue a command in the following form:

```
mongod --config /etc/mongod.conf
mongod -f /etc/mongod.conf
```

Modify the values in the `/etc/mongod.conf` file on your system to control the configuration of your database instance.

Configure the Database

Consider the following basic configuration which uses the `YAML format`:

```
processManagement:
   fork: true
net:
   bindIp: 127.0.0.1
   port: 27017
storage:
   dbPath: /srv/mongodb
systemLog:
   destination: file
   path: "/var/log/mongodb/mongod.log"
   logAppend: true
storage:
   journal:
      enabled: true
```

Or, if using the older `.ini` configuration file format:

```
fork = true
bind_ip = 127.0.0.1
port = 27017
quiet = true
dbpath = /srv/mongodb
logpath = /var/log/mongodb/mongod.log
logappend = true
journal = true
```

For most standalone servers, this is a sufficient base configuration. It makes several assumptions, but consider the following explanation:

- `fork` is `true`, which enables a *daemon* mode for `mongod`, which detaches (i.e. "forks") the MongoDB from the current session and allows you to run the database as a conventional server.
- `bindIp` is `127.0.0.1`, which forces the server to only listen for requests on the localhost IP. Only bind to secure interfaces that the application-level systems can access with access control provided by system network filtering (i.e. "*firewall*").

  New in version 2.6: `mongod` installed from official `.deb` and `.rpm` packages have the `bind_ip` configuration set to `127.0.0.1` by default.
- `port` is `27017`, which is the default MongoDB port for database instances. MongoDB can bind to any port. You can also filter access based on port using network filtering tools.

  Note: UNIX-like systems require superuser privileges to attach processes to ports lower than 1024.
- `quiet` is `true`. This disables all but the most critical entries in output/log file, and is *not* recommended for production systems. If you do set this option, you can use `setParameter` to modify this setting during run time.
- `dbPath` is `/srv/mongodb`, which specifies where MongoDB will store its data files. `/srv/mongodb` and `/var/lib/mongodb` are popular locations. The user account that `mongod` runs under will need read and write access to this directory.
- `systemLog.path` is `/var/log/mongodb/mongod.log` which is where `mongod` will write its output. If you do not set this value, `mongod` writes all output to standard output (e.g. `stdout`.)
- `logAppend` is `true`, which ensures that `mongod` does not overwrite an existing log file following the server start operation.
- `storage.journal.enabled` is `true`, which enables *journaling*. Journaling ensures single instance write-durability. 64-bit builds of `mongod` enable journaling by default. Thus, this setting may be redundant.

Given the default configuration, some of these values may be redundant. However, in many situations explicitly stating the configuration increases overall system intelligibility.

Security Considerations

The following collection of configuration options are useful for limiting access to a `mongod` instance. Consider the following settings, shown in both `YAML` and older configuration file format:

In `YAML format`

```
security:
   authorization: enabled
net:
   bindIp: 127.0.0.1,10.8.0.10,192.168.4.24
```

Or, if using the older older configuration file format[46]:

```
bind_ip = 127.0.0.1,10.8.0.10,192.168.4.24
auth = true
```

Consider the following explanation for these configuration decisions:

[46]https://docs.mongodb.org/v2.4/reference/configuration-options

- "`bindIp`" has three values: `127.0.0.1`, the localhost interface; `10.8.0.10`, a private IP address typically used for local networks and VPN interfaces; and `192.168.4.24`, a private network interface typically used for local networks.

  Because production MongoDB instances need to be accessible from multiple database servers, it is important to bind MongoDB to multiple interfaces that are accessible from your application servers. At the same time it's important to limit these interfaces to interfaces controlled and protected at the network layer.

- "`authorization`" is `true` enables the authorization system within MongoDB. If enabled you will need to log in by connecting over the `localhost` interface for the first time to create user credentials.

See also:

`https://docs.mongodb.org/manual/security`

Replication and Sharding Configuration

Replication Configuration

Replica set configuration is straightforward, and only requires that the `replSetName` have a value that is consistent among all members of the set. Consider the following:

In `YAML format`

```
replication:
   replSetName: set0
```

Or, if using the older configuration file format[47]:

```
replSet = set0
```

Use descriptive names for sets. Once configured, use the `mongo` shell to add hosts to the replica set.

See also:

Replica set reconfiguration.

To enable authentication for the *replica set*, add the following `keyFile` option:

In `YAML format`

```
security:
   keyFile: /srv/mongodb/keyfile
```

Or, if using the older configuration file format[48]:

```
keyFile = /srv/mongodb/keyfile
```

Setting `keyFile` enables authentication and specifies a key file for the replica set member use to when authenticating to each other. The content of the key file is arbitrary, but must be the same on all members of the *replica set* and `mongos` instances that connect to the set. The keyfile must be less than one kilobyte in size and may only contain characters in the base64 set and the file must not have group or "world" permissions on UNIX systems.

See also:

The *Replica Set Security* section for information on configuring authentication with replica sets.

The `https://docs.mongodb.org/manual/replication` document for more information on replication in MongoDB and replica set configuration in general.

[47] https://docs.mongodb.org/v2.4/reference/configuration-options
[48] https://docs.mongodb.org/v2.4/reference/configuration-options

Sharding Configuration

Sharding requires a number of `mongod` instances with different configurations. The config servers store the cluster's metadata, while the cluster distributes data among one or more shard servers.

Note: *Config servers* are not *replica sets*.

To set up one or three "config server" instances as *normal* (page 13) `mongod` instances, and then add the following configuration option:

In `YAML format`

```
sharding:
   clusterRole: configsvr
net:
   bindIp: 10.8.0.12
   port: 27001
```

Or, if using the older configuration file format[49]:

```
configsvr = true

bind_ip = 10.8.0.12
port = 27001
```

This creates a config server running on the private IP address `10.8.0.12` on port `27001`. Make sure that there are no port conflicts, and that your config server is accessible from all of your `mongos` and `mongod` instances.

To set up shards, configure two or more `mongod` instance using your *base configuration* (page 13), with the `shardsvr` value for the `sharding.clusterRole` setting:

```
sharding:
   clusterRole: shardsvr
```

Or, if using the older configuration file format[50]:

```
shardsvr = true
```

Finally, to establish the cluster, configure at least one `mongos` process with the following settings:

In `YAML format:`

```
sharding:
   configDB: 10.8.0.12:27001
   chunkSize: 64
```

Or, if using the older configuration file format[51]:

```
configdb = 10.8.0.12:27001
chunkSize = 64
```

Important: Always use 3 config servers in production environments.

You can specify multiple `configDB` instances by specifying hostnames and ports in the form of a comma separated list.

[49] https://docs.mongodb.org/v2.4/reference/configuration-options
[50] https://docs.mongodb.org/v2.4/reference/configuration-options
[51] https://docs.mongodb.org/v2.4/reference/configuration-options

In general, avoid modifying the `chunkSize` from the default value of 64, [52] and ensure this setting is consistent among all `mongos` instances.

See also:

The `https://docs.mongodb.org/manual/sharding` section of the manual for more information on sharding and cluster configuration.

Run Multiple Database Instances on the Same System

In many cases running multiple instances of `mongod` on a single system is not recommended. On some types of deployments [53] and for testing purposes you may need to run more than one `mongod` on a single system.

In these cases, use a *base configuration* (page 13) for each instance, but consider the following configuration values:

In `YAML format`:

```
storage:
   dbPath: /srv/mongodb/db0/
processManagement:
   pidFilePath: /srv/mongodb/db0.pid
```

Or, if using the older configuration file format[54]:

```
dbpath = /srv/mongodb/db0/
pidfilepath = /srv/mongodb/db0.pid
```

The `dbPath` value controls the location of the `mongod` instance's data directory. Ensure that each database has a distinct and well labeled data directory. The `pidFilePath` controls where `mongod` process places it's *process id* file. As this tracks the specific `mongod` file, it is crucial that file be unique and well labeled to make it easy to start and stop these processes.

Create additional *init scripts* and/or adjust your existing MongoDB configuration and init script as needed to control these processes.

Diagnostic Configurations

The following configuration options control various `mongod` behaviors for diagnostic purposes:

- `operationProfiling.mode` sets the *database profiler* (page 39) level. The profiler is not active by default because of the possible impact on the profiler itself on performance. Unless this setting is on, queries are not profiled.
- `operationProfiling.slowOpThresholdMs` configures the threshold which determines whether a query is "slow" for the purpose of the logging system and the *profiler* (page 39). The default value is 100 milliseconds. Set a lower value if the database profiler does not return useful results or a higher value to only log the longest running queries.
- `systemLog.verbosity` controls the amount of logging output that `mongod` write to the log. Only use this option if you are experiencing an issue that is not reflected in the normal logging level.

[52] *Chunk* size is 64 megabytes by default, which provides the ideal balance between the most even distribution of data, for which smaller chunk sizes are best, and minimizing chunk migration, for which larger chunk sizes are optimal.

[53] Single-tenant systems with *SSD* or other high performance disks may provide acceptable performance levels for multiple `mongod` instances. Additionally, you may find that multiple databases with small working sets may function acceptably on a single system.

[54] https://docs.mongodb.org/v2.4/reference/configuration-options

Changed in version 3.0: You can also specify verbosity level for specific components using the `systemLog.component.<name>.verbosity` setting. For the available components, see `component verbosity settings`.

For more information, see also *Database Profiling* (page 39) and *Analyzing MongoDB Performance* (page 37).

1.1.4 Production Notes

On this page

This page details system configurations that affect MongoDB, especially in production.

Note: MongoDB Cloud Manager[55], a hosted service, and Ops Manager[56], an on-premise solution, provide monitoring, backup, and automation of MongoDB instances. See the MongoDB Cloud Manager documentation[57] and Ops Manager documentation[58] for more information.

MongoDB Binaries

Supported Platforms

Platform	3.0	2.6	2.4	2.2
Amazon Linux	Y	Y	Y	Y
Debian 7	Y	Y	Y	Y
Fedora 8+		Y	Y	Y
RHEL/CentOS 6.2+	Y	Y	Y	Y
RHEL/CentOS 7.0+	Y	Y		
SLES 11+	Y	Y	Y	Y
Solaris 64-bit	Y	Y	Y	Y
Ubuntu 12.04	Y	Y	Y	Y
Ubuntu 14.04	Y	Y		
Microsoft Azure	Y	Y	Y	Y
Windows Vista/Server 2008R2/2012+	Y	Y	Y	Y
OSX 10.7+	Y	Y	Y	

[55] https://cloud.mongodb.com/?jmp=docs
[56] https://www.mongodb.com/products/mongodb-enterprise-advanced?jmp=docs
[57] https://docs.cloud.mongodb.com/
[58] https://docs.opsmanager.mongodb.com?jmp=docs

Note: The MongoDB distribution for Solaris does not include support for the *WiredTiger storage engine*.

Recommended Platforms

We recommend the following operating systems for production use:

- Amazon Linux
- Debian 7.1
- Red Hat / CentOS 6.2+
- SLES 11+
- Ubuntu LTS 12.04
- Ubuntu LTS 14.04
- Windows Server 2012 & 2012 R2

See also:

Platform Specific Considerations (page 25)

Use the Latest Stable Packages

Be sure you have the latest stable release.

All releases are available on the Downloads[59] page. The Downloads[60] page is a good place to verify the current stable release, even if you are installing via a package manager.

Use 64-bit Builds

Always use 64-bit builds for production.

Although the 32-bit builds exist, they are **unsuitable** for production deployments. 32-bit builds also do **not** support the WiredTiger storage engine. For more information, see the *32-bit limitations page*

MongoDB `dbPath`

Changed in version 3.0: MongoDB includes support for two storage engines: *MMAPv1*, the storage engine available in previous versions of MongoDB, and *WiredTiger*. MongoDB uses the MMAPv1 engine by default.

The files in the `dbPath` directory must correspond to the configured storage engine. `mongod` will not start if `dbPath` contains data files created by a storage engine other than the one specified by `--storageEngine`.

Concurrency

MMAPv1

Changed in version 3.0: Beginning with MongoDB 3.0, *MMAPv1* provides *collection-level locking*: All collections have a unique readers-writer lock that allows multiple clients to modify documents in different collections at the same time.

[59] http://www.mongodb.org/downloads
[60] http://www.mongodb.org/downloads

For MongoDB versions 2.2 through 2.6 series, each database has a readers-writer lock that allows concurrent read access to a database, but gives exclusive access to a single write operation per database. See the `Concurrency` page for more information. In earlier versions of MongoDB, all write operations contended for a single readers-writer lock for the entire `mongod` instance.

WiredTiger

WiredTiger supports concurrent access by readers and writers to the documents in a collection. Clients can read documents while write operations are in progress, and multiple threads can modify different documents in a collection at the same time.

See also:

Allocate Sufficient RAM and CPU (page 21)

Data Consistency

Journaling

MongoDB uses *write ahead logging* to an on-disk *journal*. Journaling guarantees that MongoDB can quickly recover `write operations` that were written to the journal but not written to data files in cases where `mongod` terminated as a result of a crash or other serious failure.

Leave journaling enabled in order to ensure that `mongod` will be able to recover its data files and keep the data files in a valid state following a crash. See *Journaling* (page 147) for more information.

Write Concern

`Write concern` describes the level of acknowledgement requested from MongoDB for write operations. The level of the write concerns affects how quickly the write operation returns. When write operations have a *weak* write concern, they return quickly. With *stronger* write concerns, clients must wait after sending a write operation until MongoDB confirms the write operation at the requested write concern level. With insufficient write concerns, write operations may appear to a client to have succeeded, but may not persist in some cases of server failure.

See the `Write Concern` document for more information about choosing an appropriate write concern level for your deployment.

Networking

Use Trusted Networking Environments

Always run MongoDB in a *trusted environment*, with network rules that prevent access from *all* unknown machines, systems, and networks. As with any sensitive system that is dependent on network access, your MongoDB deployment should only be accessible to specific systems that require access, such as application servers, monitoring services, and other MongoDB components.

Note: By default, `authorization` is not enabled, and `mongod` assumes a trusted environment. Enable `authorization` mode as needed. For more information on authentication mechanisms supported in MongoDB as well as authorization in MongoDB, see `https://docs.mongodb.org/manual/core/authentication` and `https://docs.mongodb.org/manual/core/authorization`.

For additional information and considerations on security, refer to the documents in the `Security Section`, specifically:

- `https://docs.mongodb.org/manual/administration/security-checklist`
- `https://docs.mongodb.org/manual/core/security-mongodb-configuration`
- `https://docs.mongodb.org/manual/core/security-network`
- `Network Security Tutorials`

For Windows users, consider the Windows Server Technet Article on TCP Configuration[61] when deploying MongoDB on Windows.

Disable HTTP Interfaces

MongoDB provides interfaces to check the status of the server and, optionally, run queries on it, over HTTP. In production environments, disable the HTTP interfaces.

See *http-interface-security*.

Manage Connection Pool Sizes

To avoid overloading the connection resources of a single `mongod` or `mongos` instance, ensure that clients maintain reasonable connection pool sizes. Adjust the connection pool size to suit your use case, beginning at 110-115% of the typical number of concurrent database requests.

The `connPoolStats` command returns information regarding the number of open connections to the current database for `mongos` and `mongod` instances in sharded clusters.

See also *Allocate Sufficient RAM and CPU* (page 21).

Hardware Considerations

MongoDB is designed specifically with commodity hardware in mind and has few hardware requirements or limitations. MongoDB's core components run on little-endian hardware, primarily x86/x86_64 processors. Client libraries (i.e. drivers) can run on big or little endian systems.

Allocate Sufficient RAM and CPU

MMAPv1 Due to its concurrency model, the MMAPv1 storage engine does not require many CPU cores . As such, increasing the number of cores can help but does not provide significant return.

Increasing the amount of RAM accessible to MongoDB may help reduce the frequency of page faults.

WiredTiger The WiredTiger storage engine is multithreaded and can take advantage of many CPU cores. Specifically, the total number of active threads (i.e. concurrent operations) relative to the number of CPUs can impact performance:

- Throughput *increases* as the number of concurrent active operations increases up to the number of CPUs.
- Throughput *decreases* as the number of concurrent active operations exceeds the number of CPUs by some threshold amount.

[61] http://technet.microsoft.com/en-us/library/dd349797.aspx

The threshold amount depends on your application. You can determine the optimum number of concurrent active operations for your application by experimenting and measuring throughput. The output from `mongostat` provides statistics on the number of active reads/writes in the (`ar|aw`) column.

By default, the WiredTiger cache will use either 1GB or half of the installed physical RAM, whichever is larger.

The size of the cache is tunable through the `storage.wiredTiger.engineConfig.cacheSizeGB` setting. If the cache does not have enough space to load additional data, WiredTiger evicts pages from the cache to free up space.

Note: The `storage.wiredTiger.engineConfig.cacheSizeGB` only limits the size of the WiredTiger cache, not the total amount of memory used by `mongod`. The WiredTiger cache is only one component of the RAM used by MongoDB. MongoDB also automatically uses all free memory on the machine via the filesystem cache (data in the filesystem cache is compressed).

In addition, the operating system will use any free RAM to buffer filesystem blocks.

To accommodate the additional consumers of RAM, you may have to decrease WiredTiger cache size. Avoid increasing the WiredTiger cache size above its default value.

The default WiredTiger cache size value assumes that there is a single `mongod` instance per node. If a single node contains multiple instances, then you should decrease the setting to accommodate the other `mongod` instances.

If you run `mongod` in a container (e.g. `lxc`, `cgroups`, Docker, etc.) that does *not* have access to all of the RAM available in a system, you must set `storage.wiredTiger.engineConfig.cacheSizeGB` to a value less than the amount of RAM available in the container. The exact amount depends on the other processes running in the container.

To view statistics on the cache and eviction rate, see the `wiredTiger.cache` field returned from the `serverStatus` command.

To adjust the size of the WiredTiger cache, see `storage.wiredTiger.engineConfig.cacheSizeGB` and `--wiredTigerCacheSizeGB`

See also:

Concurrency (page 19)

Use Solid State Disks (SSDs)

MongoDB has good results and a good price-performance ratio with SATA SSD (Solid State Disk).

Use SSD if available and economical. Spinning disks can be performant, but SSDs' capacity for random I/O operations works well with the update model of MMAPv1.

Commodity (SATA) spinning drives are often a good option, as the random I/O performance increase with more expensive spinning drives is not that dramatic (only on the order of 2x). Using SSDs or increasing RAM may be more effective in increasing I/O throughput.

MongoDB and NUMA Hardware

Running MongoDB on a system with Non-Uniform Access Memory (NUMA) can cause a number of operational problems, including slow performance for periods of time and high system process usage.

When running MongoDB servers and clients on NUMA hardware, you should configure a memory interleave policy so that the host behaves in a non-NUMA fashion. MongoDB checks NUMA settings on start up when deployed on Linux (since version 2.0) and Windows (since version 2.6) machines. If the NUMA configuration may degrade performance, MongoDB prints a warning.

See also:

- The MySQL "swap insanity" problem and the effects of NUMA[62] post, which describes the effects of NUMA on databases. The post introduces NUMA and its goals, and illustrates how these goals are not compatible with production databases. Although the blog post addresses the impact of NUMA for MySQL, the issues for MongoDB are similar.
- NUMA: An Overview[63].

Configuring NUMA on Windows On Windows, memory interleaving must be enabled through the machine's BIOS. Please consult your system documentation for details.

Configuring NUMA on Linux When running MongoDB on Linux, you may instead use the `numactl` command and start the MongoDB programs (`mongod`, including the `config servers`; `mongos`; or clients) in the following manner:

```
numactl --interleave=all <path>
```

where `<path>` is the path to the program you are starting. Then, disable *zone reclaim* in the `proc` settings using the following command:

```
echo 0 > /proc/sys/vm/zone_reclaim_mode
```

To fully disable NUMA behavior, you must perform both operations. For more information, see the Documentation for /proc/sys/vm/*[64].

Disk and Storage Systems

Swap Assign swap space for your systems. Allocating swap space can avoid issues with memory contention and can prevent the OOM Killer on Linux systems from killing `mongod`.

For the MMAPv1 storage engine, the method `mongod` uses to map files to memory ensures that the operating system will never store MongoDB data in swap space. On Windows systems, using MMAPv1 requires extra swap space due to commitment limits. For details, see *MongoDB on Windows* (page 27).

For the WiredTiger storage engine, given sufficient memory pressure, WiredTiger may store data in swap space .

RAID Most MongoDB deployments should use disks backed by RAID-10.

RAID-5 and RAID-6 do not typically provide sufficient performance to support a MongoDB deployment.

Avoid RAID-0 with MongoDB deployments. While RAID-0 provides good write performance, it also provides limited availability and can lead to reduced performance on read operations, particularly when using Amazon's EBS volumes.

[62] http://jcole.us/blog/archives/2010/09/28/mysql-swap-insanity-and-the-numa-architecture/
[63] https://queue.acm.org/detail.cfm?id=2513149
[64] http://www.kernel.org/doc/Documentation/sysctl/vm.txt

Remote Filesystems With the MMAPv1 storage engine, the Network File System protocol (NFS) is not recommended as you may see performance problems when both the data files and the journal files are hosted on NFS. You may experience better performance if you place the journal on local or `iscsi` volumes.

With the WiredTiger storage engine, WiredTiger objects may be stored on remote file systems if the remote file system conforms to ISO/IEC 9945-1:1996 (POSIX.1). Because remote file systems are often slower than local file systems, using a remote file system for storage may degrade performance.

If you decide to use NFS, add the following NFS options to your `/etc/fstab` file: `bg`, `nolock`, and `noatime`.

Separate Components onto Different Storage Devices For improved performance, consider separating your database's data, journal, and logs onto different storage devices, based on your application's access and write pattern.

For the WiredTiger storage engine, you can also store the indexes on a different storage device. See `storage.wiredTiger.engineConfig.directoryForIndexes`.

Note: Using different storage devices will affect your ability to create snapshot-style backups of your data, since the files will be on different devices and volumes.

Scheduling for Virtual Devices Local block devices attached to virtual machine instances via the hypervisor should use a *noop* scheduler for best performance. The *noop* scheduler allows the operating system to defer I/O scheduling to the underlying hypervisor.

Architecture

Replica Sets

See the `Replica Set Architectures` document for an overview of architectural considerations for replica set deployments.

Sharded Clusters

See the `Sharded Cluster Production Architecture` document for an overview of recommended sharded cluster architectures for production deployments.

See also:

Design Notes (page 43)

Compression

WiredTiger can compress collection data using either *snappy* or *zlib* compression library. *snappy* provides a lower compression rate but has little performance cost, whereas `zlib` provides better compression rate but has a higher performance cost.

By default, WiredTiger uses *snappy* compression library. To change the compression setting, see `storage.wiredTiger.collectionConfig.blockCompressor`.

WiredTiger uses *prefix compression* on all indexes by default.

Platform Specific Considerations

Note: MongoDB uses the GNU C Library[65] (glibc) if available on a system. MongoDB requires version at least `glibc-2.12-1.2.el6` to avoid a known bug with earlier versions. For best results use at least version 2.13.

MongoDB on Linux

Kernel and File Systems When running MongoDB in production on Linux, it is recommended that you use Linux kernel version 2.6.36 or later.

With the MMAPv1 storage engine, MongoDB preallocates its database files before using them and often creates large files. As such, you should use the XFS and EXT4 file systems. If possible, use XFS as it generally performs better with MongoDB.

With the WiredTiger storage engine, use of XFS is **strongly recommended** to avoid performance issues that have been observed when using EXT4 with WiredTiger.

- In general, if you use the XFS file system, use at least version `2.6.25` of the Linux Kernel.
- In general, if you use the EXT4 file system, use at least version `2.6.23` of the Linux Kernel.
- Some Linux distributions require different versions of the kernel to support using XFS and/or EXT4:

Linux Distribution	Filesystem	Kernel Version
CentOS 5.5	ext4, xfs	`2.6.18-194.el5`
CentOS 5.6	ext4, xfs	`2.6.18-3.0.el5`
CentOS 5.8	ext4, xfs	`2.6.18-308.8.2.el5`
CentOS 6.1	ext4, xfs	`2.6.32-131.0.15.el6.x86_64`
RHEL 5.6	ext4	`2.6.18-3.0`
RHEL 6.0	xfs	`2.6.32-71`
Ubuntu 10.04.4 LTS	ext4, xfs	`2.6.32-38-server`
Amazon Linux AMI release 2012.03	ext4	`3.2.12-3.2.4.amzn1.x86_64`

`fsync()` on Directories

Important: MongoDB requires a filesystem that supports `fsync()` *on directories*. For example, HGFS and Virtual Box's shared folders do *not* support this operation.

Recommended Configuration For the MMAPv1 storage engine and the WiredTiger storage engines, consider the following recommendations:

- Turn off `atime` for the storage volume containing the *database files*.
- Set the file descriptor limit, `-n`, and the user process limit (ulimit), `-u`, above 20,000, according to the suggestions in the *ulimit* (page 125) document. A low ulimit will affect MongoDB when under heavy use and can produce errors and lead to failed connections to MongoDB processes and loss of service.
- Disable Transparent Huge Pages, as MongoDB performs better with normal (4096 bytes) virtual memory pages. See *Transparent Huge Pages Settings* (page 48).
- Disable NUMA in your BIOS. If that is not possible, see *MongoDB on NUMA Hardware* (page 22).
- Configure SELinux on Red Hat. For more information, see *Configure SELinux for MongoDB* and *Configure SELinux for MongoDB Enterprise*.

[65] http://www.gnu.org/software/libc/

For the MMAPv1 storage engine:

- Ensure that readahead settings for the block devices that store the database files are appropriate. For random access use patterns, set low readahead values. A readahead of 32 (16kb) often works well.

  For a standard block device, you can run `sudo blockdev --report` to get the readahead settings and `sudo blockdev --setra <value> <device>` to change the readahead settings. Refer to your specific operating system manual for more information.

For all MongoDB deployments:

- Use the Network Time Protocol (NTP) to synchronize time among your hosts. This is especially important in sharded clusters.

MongoDB and TLS/SSL Libraries On Linux platforms, you may observe one of the following statements in the MongoDB log:

```
<path to SSL libs>/libssl.so.<version>: no version information available (required by /usr/bin/m
<path to SSL libs>/libcrypto.so.<version>: no version information available (required by /usr/bi
```

These warnings indicate that the system's TLS/SSL libraries are different from the TLS/SSL libraries that the `mongod` was compiled against. Typically these messages do not require intervention; however, you can use the following operations to determine the symbol versions that `mongod` expects:

```
objdump -T <path to mongod>/mongod | grep " SSL_"
objdump -T <path to mongod>/mongod | grep " CRYPTO_"
```

These operations will return output that resembles one the of the following lines:

```
0000000000000000      DF *UND*       0000000000000000  libssl.so.10 SSL_write
0000000000000000      DF *UND*       0000000000000000  OPENSSL_1.0.0 SSL_write
```

The last two strings in this output are the symbol version and symbol name. Compare these values with the values returned by the following operations to detect symbol version mismatches:

```
objdump -T <path to TLS/SSL libs>/libssl.so.1*
objdump -T <path to TLS/SSL libs>/libcrypto.so.1*
```

This procedure is neither exact nor exhaustive: many symbols used by `mongod` from the `libcrypto` library do not begin with `CRYPTO_`.

MongoDB on Windows

MongoDB Using MMAPv1

Install Hotfix for MongoDB 2.6.6 and Later Microsoft has released a hotfix for Windows 7 and Windows Server 2008 R2, KB2731284[66], that repairs a bug in these operating systems' use of memory-mapped files that adversely affects the performance of MongoDB using the MMAPv1 storage engine.

Install this hotfix to obtain significant performance improvements on MongoDB 2.6.6 and later releases in the 2.6 series, which use MMAPv1 exclusively, and on 3.0 and later when using MMAPv1 as the storage engine.

66 http://support.microsoft.com/kb/2731284

Configure Windows Page File For MMAPv1 Configure the page file such that the minimum and maximum page file size are equal and at least 32 GB. Use a multiple of this size if, during peak usage, you expect concurrent writes to many databases or collections. However, the page file size does not need to exceed the maximum size of the database.

A large page file is needed as Windows requires enough space to accommodate all regions of memory mapped files made writable during peak usage, regardless of whether writes actually occur.

The page file is not used for database storage and will not receive writes during normal MongoDB operation. As such, the page file will not affect performance, but it must exist and be large enough to accommodate Windows' commitment rules during peak database use.

Note: Dynamic page file sizing is too slow to accommodate the rapidly fluctuating commit charge of an active MongoDB deployment. This can result in transient overcommitment situations that may lead to abrupt server shutdown with a VirtualProtect error 1455.

MongoDB 3.0 Using WiredTiger For MongoDB instances using the WiredTiger storage engine, performance on Windows is comparable to performance on Linux.

MongoDB on Virtual Environments

This section describes considerations when running MongoDB in some of the more common virtual environments.

For all platforms, consider *Scheduling for Virtual Devices* (page 24).

EC2 MongoDB is compatible with EC2. MongoDB Cloud Manager[67] provides integration with Amazon Web Services (AWS) and lets you deploy new EC2 instances directly from MongoDB Cloud Manager. See Configure AWS Integration[68] for more details.

Azure For all MongoDB deployments using Azure, you **must** mount the volume that hosts the `mongod` instance's `dbPath` with the *Host Cache Preference* `READ/WRITE`.

This applies to all Azure deployments, using any guest operating system.

If your volumes have inappropriate cache settings, MongoDB may eventually shut down with the following error:

```
[DataFileSync] FlushViewOfFile for <data file> failed with error 1 ...
[DataFileSync] Fatal Assertion 16387
```

These shut downs do not produce data loss when `storage.journal.enabled` is set to `true`. You can safely restart `mongod` at any time following this event.

The performance characteristics of MongoDB may change with `READ/WRITE` caching enabled.

The TCP keepalive on the Azure load balancer is 240 seconds by default, which can cause it to silently drop connections if the TCP keepalive on your Azure systems is greater than this value. You should set `tcp_keepalive_time` to 120 to ameliorate this problem.

On Linux systems:

- To view the keep alive setting, you can use one of the following commands:

[67] https://cloud.mongodb.com/?jmp=docs

[68] https://docs.cloud.mongodb.com/tutorial/configure-aws-settings/

```
sysctl net.ipv4.tcp_keepalive_time
```

Or:

```
cat /proc/sys/net/ipv4/tcp_keepalive_time
```

The value is measured in seconds.

- To change the `tcp_keepalive_time` value, you can use one of the following command:

```
sudo sysctl -w net.ipv4.tcp_keepalive_time=<value>
```

Or:

```
echo <value> | sudo tee /proc/sys/net/ipv4/tcp_keepalive_time
```

These operations do not persist across system reboots. To persist the setting, add the following line to `/etc/sysctl.conf`:

```
net.ipv4.tcp_keepalive_time = <value>
```

On Linux, `mongod` and `mongos` processes limit the keepalive to a maximum of 300 seconds (5 minutes) on their own sockets by overriding keepalive values greater than 5 minutes.

For Windows systems:

- To view the keep alive setting, issue the following command:

```
reg query HKLM\SYSTEM\CurrentControlSet\Services\Tcpip\Parameters /v KeepAliveTime
```

The registry value is not present by default. The system default, used if the value is absent, is 7200000 *milliseconds* or `0x6ddd00` in hexadecimal.

- To change the `KeepAliveTime` value, use the following command in an Administrator *Command Prompt*, where `<value>` is expressed in hexadecimal (e.g. `0x0124c0` is 120000):

```
reg add HKLM\SYSTEM\CurrentControlSet\Services\Tcpip\Parameters\ /v KeepAliveTime /d <value>
```

Windows users should consider the Windows Server Technet Article on KeepAliveTime[69] for more information on setting keep alive for MongoDB deployments on Windows systems.

VMWare MongoDB is compatible with VMWare.

As some users have run into issues with VMWare's memory overcommit feature, you should disable the feature.

Further, MongoDB is known to run poorly with VMWare's balloon driver (`vmmemctl`), so you should disable this as well. VMWare uses the balloon driver to reduce physical memory usage on the host hardware by allowing the hypervisor to swap to disk while hiding this fact from the guest, which continues to see the same amount of (virtual) physical memory. This interferes with MongoDB's memory management, and you are likely to experience significant performance degradation.

It is possible to clone a virtual machine running MongoDB. You might use this function to spin up a new virtual host to add as a member of a replica set. If you clone a VM with journaling enabled, the clone snapshot will be valid. If not using journaling, first stop `mongod`, then clone the VM, and finally, restart `mongod`.

[69] https://technet.microsoft.com/en-us/library/cc957549.aspx

Performance Monitoring

iostat

On Linux, use the `iostat` command to check if disk I/O is a bottleneck for your database. Specify a number of seconds when running iostat to avoid displaying stats covering the time since server boot.

For example, the following command will display extended statistics and the time for each displayed report, with traffic in MB/s, at one second intervals:

```
iostat -xmt 1
```

Key fields from `iostat`:

- `%util`: this is the most useful field for a quick check, it indicates what percent of the time the device/drive is in use.
- `avgrq-sz`: average request size. Smaller number for this value reflect more random IO operations.

bwm-ng

bwm-ng[70] is a command-line tool for monitoring network use. If you suspect a network-based bottleneck, you may use `bwm-ng` to begin your diagnostic process.

Backups

To make backups of your MongoDB database, please refer to *MongoDB Backup Methods Overview* (page 4).

Additional Resources

- Blog Post: Capacity Planning and Hardware Provisioning for MongoDB In Ten Minutes[71]
- Whitepaper: MongoDB Multi-Data Center Deployments[72]
- Whitepaper: Security Architecture[73]
- Whitepaper: MongoDB Architecture Guide[74]
- Presentation: MongoDB Administration 101[75]
- MongoDB Production Readiness Consulting Package[76]

1.2 Data Management

These document introduce data management practices and strategies for MongoDB deployments, including strategies for managing multi-data center deployments, managing larger file stores, and data lifecycle tools.

[70] http://www.gropp.org/?id=projects&sub=bwm-ng
[71] https://www.mongodb.com/blog/post/capacity-planning-and-hardware-provisioning-mongodb-ten-minutes?jmp=docs
[72] http://www.mongodb.com/lp/white-paper/multi-dc?jmp=docs
[73] https://www.mongodb.com/lp/white-paper/mongodb-security-architecture?jmp=docs
[74] https://www.mongodb.com/lp/whitepaper/architecture-guide?jmp=docs
[75] http://www.mongodb.com/presentations/webinar-mongodb-administration-101?jmp=docs
[76] https://www.mongodb.com/products/consulting?jmp=docs#s_product_readiness

Data Center Awareness **(page 30)** Presents the MongoDB features that allow application developers and database administrators to configure their deployments to be more data center aware or allow operational and location-based separation.

Capped Collections **(page 32)** Capped collections provide a special type of size-constrained collections that preserve insertion order and can support high volume inserts.

Expire Data from Collections by Setting TTL **(page 35)** TTL collections make it possible to automatically remove data from a collection based on the value of a timestamp and are useful for managing data like machine generated event data that are only useful for a limited period of time.

1.2.1 Data Center Awareness

On this page

MongoDB provides a number of features that allow application developers and database administrators to customize the behavior of a *sharded cluster* or *replica set* deployment so that MongoDB may be *more* "data center aware," or allow operational and location-based separation.

MongoDB also supports segregation based on functional parameters, to ensure that certain `mongod` instances are only used for reporting workloads or that certain high-frequency portions of a sharded collection only exist on specific shards.

The following documents, *found either in this section or other sections of this manual*, provide information on customizing a deployment for operation- and location-based separation:

Operational Segregation in MongoDB Deployments **(page 30)** MongoDB lets you specify that certain application operations use certain `mongod` instances.

`https://docs.mongodb.org/manual/core/tag-aware-sharding` Tags associate specific ranges of *shard key* values with specific shards for use in managing deployment patterns.

Manage Shard Tags **(page 254)** Use tags to associate specific ranges of shard key values with specific shards.

Operational Segregation in MongoDB Deployments

On this page

Operational Overview

MongoDB includes a number of features that allow database administrators and developers to segregate application operations to MongoDB deployments by functional or geographical groupings.

This capability provides "data center awareness," which allows applications to target MongoDB deployments with consideration of the physical location of the `mongod` instances. MongoDB supports segmentation of operations across different dimensions, which may include multiple data centers and geographical regions in multi-data center deployments, racks, networks, or power circuits in single data center deployments.

MongoDB also supports segregation of database operations based on functional or operational parameters, to ensure that certain `mongod` instances are only used for reporting workloads or that certain high-frequency portions of a sharded collection only exist on specific shards.

Specifically, with MongoDB, you can:

- ensure write operations propagate to specific members of a replica set, or to specific members of replica sets.
- ensure that specific members of a replica set respond to queries.
- ensure that specific ranges of your *shard key* balance onto and reside on specific *shards*.
- combine the above features in a single distributed deployment, on a per-operation (for read and write operations) and collection (for chunk distribution in sharded clusters distribution) basis.

For full documentation of these features, see the following documentation in the MongoDB Manual:

- `Read Preferences`, which controls how drivers help applications target read operations to members of a replica set.
- `Write Concerns`, which controls how MongoDB ensures that write operations propagate to members of a replica set.
- *Replica Set Tags* (page 194), which control how applications create and interact with custom groupings of replica set members to create custom application-specific read preferences and write concerns.
- *Tag Aware Sharding*, which allows MongoDB administrators to define an application-specific balancing policy, to control how documents belonging to specific ranges of a shard key distribute to shards in the *sharded cluster*.

See also:

Before adding operational segregation features to your application and MongoDB deployment, become familiar with all documentation of `replication`, and `sharding`.

Additional Resource

- Whitepaper: MongoDB Multi-Data Center Deployments[77]
- Webinar: Multi-Data Center Deployment[78]

Further Reading

- The `https://docs.mongodb.org/manual/reference/write-concern` and `https://docs.mongodb.org/manual/core/read-preference` documents, which address capabilities related to data center awareness.
- *Deploy a Geographically Redundant Replica Set* (page 165).

Additional Resource

- Whitepaper: MongoDB Multi-Data Center Deployments[79]

[77] http://www.mongodb.com/lp/white-paper/multi-dc?jmp=docs
[78] https://www.mongodb.com/presentations/webinar-multi-data-center-deployment?jmp=docs
[79] http://www.mongodb.com/lp/white-paper/multi-dc?jmp=docs

- Webinar: Multi-Data Center Deployment[80]

1.2.2 Capped Collections

On this page

Overview

Capped collections are fixed-size collections that support high-throughput operations that insert and retrieve documents based on insertion order. Capped collections work in a way similar to circular buffers: once a collection fills its allocated space, it makes room for new documents by overwriting the oldest documents in the collection.

See `createCollection()` or `create` for more information on creating capped collections.

Behavior

Insertion Order

Capped collections guarantee preservation of the insertion order. As a result, queries do not need an index to return documents in insertion order. Without this indexing overhead, capped collections can support higher insertion throughput.

Automatic Removal of Oldest Documents

To make room for new documents, capped collections automatically remove the oldest documents in the collection without requiring scripts or explicit remove operations.

For example, the *oplog.rs* collection that stores a log of the operations in a *replica set* uses a capped collection. Consider the following potential use cases for capped collections:

- Store log information generated by high-volume systems. Inserting documents in a capped collection without an index is close to the speed of writing log information directly to a file system. Furthermore, the built-in *first-in-first-out* property maintains the order of events, while managing storage use.
- Cache small amounts of data in a capped collections. Since caches are read rather than write heavy, you would either need to ensure that this collection *always* remains in the working set (i.e. in RAM) *or* accept some write penalty for the required index or indexes.

`_id` Index

Changed in version 2.4.

Capped collections have an `_id` field and an index on the `_id` field by default.

[80]https://www.mongodb.com/presentations/webinar-multi-data-center-deployment?jmp=docs

Restrictions and Recommendations

Updates

If you plan to update documents in a capped collection, create an index so that these update operations do not require a table scan.

With MMAPv1, you can only make in-place updates of documents. If the update operation causes a document to grow beyond the document's original size, the update operation will fail.

Replica Sets with MMAPv1 Secondaries

If you update a document in a capped collection to a size smaller than its original size and a secondary resyncs from the primary, the secondary will replicate and allocate space based on the current smaller document size.

If the primary then receives an update which increases the document back to its original size, the primary will accept the update. However, for MMAPv1, the secondary will fail with a `failing update: objects in a capped ns cannot grow` error message.

To prevent this error, create your secondary from a snapshot of one of the other up-to-date members of the replica set. Follow the :doc:‘ tutorial on filesystem snapshots </tutorial/backup-with-filesystem-snapshots>‘ to seed your new secondary.

Seeding the secondary with a filesystem snapshot is the only way to guarantee the primary and secondary binary files are compatible. MongoDB Cloud Manager Backup snapshots are insufficient in this situation since you need more than the content of the secondary to match the primary.

Document Deletion

You cannot delete documents from a capped collection. To remove all documents from a collection, use the `drop()` method to drop the collection and recreate the capped collection.

Sharding

You cannot shard a capped collection.

Query Efficiency

Use natural ordering to retrieve the most recently inserted elements from the collection efficiently. This is (somewhat) analogous to tail on a log file.

Aggregation `$out`

The aggregation pipeline operator `$out` cannot write results to a capped collection.

Procedures

Create a Capped Collection

You must create capped collections explicitly using the `createCollection()` method, which is a helper in the `mongo` shell for the `create` command. When creating a capped collection you must specify the maximum size of the collection in bytes, which MongoDB will pre-allocate for the collection. The size of the capped collection includes a small amount of space for internal overhead.

```
db.createCollection( "log", { capped: true, size: 100000 } )
```

If the `size` field is less than or equal to 4096, then the collection will have a cap of 4096 bytes. Otherwise, MongoDB will raise the provided size to make it an integer multiple of 256.

Additionally, you may also specify a maximum number of documents for the collection using the `max` field as in the following document:

```
db.createCollection("log", { capped : true, size : 5242880, max : 5000 } )
```

Important: The `size` argument is *always* required, even when you specify `max` number of documents. MongoDB will remove older documents if a collection reaches the maximum size limit before it reaches the maximum document count.

See

`createCollection()` and `create`.

Query a Capped Collection

If you perform a `find()` on a capped collection with no ordering specified, MongoDB guarantees that the ordering of results is the same as the insertion order.

To retrieve documents in reverse insertion order, issue `find()` along with the `sort()` method with the `$natural` parameter set to `-1`, as shown in the following example:

```
db.cappedCollection.find().sort( { $natural: -1 } )
```

Check if a Collection is Capped

Use the `isCapped()` method to determine if a collection is capped, as follows:

```
db.collection.isCapped()
```

Convert a Collection to Capped

You can convert a non-capped collection to a capped collection with the `convertToCapped` command:

```
db.runCommand({"convertToCapped": "mycoll", size: 100000});
```

The `size` parameter specifies the size of the capped collection in bytes.

Warning: This command obtains a global write lock and will block other operations until it has completed.

Changed in version 2.2: Before 2.2, capped collections did not have an index on `_id` unless you specified `autoIndexId` to the `create`, after 2.2 this became the default.

Automatically Remove Data After a Specified Period of Time

For additional flexibility when expiring data, consider MongoDB's *TTL* indexes, as described in *Expire Data from Collections by Setting TTL* (page 35). These indexes allow you to expire and remove data from normal collections using a special type, based on the value of a date-typed field and a TTL value for the index.

TTL Collections (page 35) are not compatible with capped collections.

Tailable Cursor

You can use a *tailable cursor* with capped collections. Similar to the Unix `tail -f` command, the tailable cursor "tails" the end of a capped collection. As new documents are inserted into the capped collection, you can use the tailable cursor to continue retrieving documents.

See `https://docs.mongodb.org/manual/tutorial/create-tailable-cursor` for information on creating a tailable cursor.

1.2.3 Expire Data from Collections by Setting TTL

On this page

New in version 2.2.

This document provides an introduction to MongoDB's "*time to live*" or *TTL* collection feature. TTL collections make it possible to store data in MongoDB and have the `mongod` automatically remove data after a specified number of seconds or at a specific clock time.

Data expiration is useful for some classes of information, including machine generated event data, logs, and session information that only need to persist for a limited period of time.

A special `TTL index property` supports the implementation of TTL collections. The TTL feature relies on a background thread in `mongod` that reads the date-typed values in the index and removes expired *documents* from the collection.

Procedures

To create a `TTL index`, use the `db.collection.createIndex()` method with the `expireAfterSeconds` option on a field whose value is either a *date* or an array that contains *date values*.

Note: The TTL index is a single field index. Compound indexes do not support the TTL property. For more information on TTL indexes, see `https://docs.mongodb.org/manual/core/index-ttl`.

Expire Documents after a Specified Number of Seconds

To expire data after a specified number of seconds has passed since the indexed field, create a TTL index on a field that holds values of BSON date type or an array of BSON date-typed objects *and* specify a positive non-zero value in the `expireAfterSeconds` field. A document will expire when the number of seconds in the `expireAfterSeconds` field has passed since the time specified in its indexed field. [81]

For example, the following operation creates an index on the `log_events` collection's `createdAt` field and specifies the `expireAfterSeconds` value of `3600` to set the expiration time to be one hour after the time specified by `createdAt`.

```
db.log_events.createIndex( { "createdAt": 1 }, { expireAfterSeconds: 3600 } )
```

When adding documents to the `log_events` collection, set the `createdAt` field to the current time:

```
db.log_events.insert( {
   "createdAt": new Date(),
   "logEvent": 2,
   "logMessage": "Success!"
} )
```

MongoDB will automatically delete documents from the `log_events` collection when the document's `createdAt` value [1] is older than the number of seconds specified in `expireAfterSeconds`.

See also:

`$currentDate` operator

Expire Documents at a Specific Clock Time

To expire documents at a specific clock time, begin by creating a TTL index on a field that holds values of BSON date type or an array of BSON date-typed objects *and* specify an `expireAfterSeconds` value of `0`. For each document in the collection, set the indexed date field to a value corresponding to the time the document should expire. If the indexed date field contains a date in the past, MongoDB considers the document expired.

For example, the following operation creates an index on the `log_events` collection's `expireAt` field and specifies the `expireAfterSeconds` value of `0`:

```
db.log_events.createIndex( { "expireAt": 1 }, { expireAfterSeconds: 0 } )
```

For each document, set the value of `expireAt` to correspond to the time the document should expire. For instance, the following `insert()` operation adds a document that should expire at `July 22, 2013 14:00:00`.

```
db.log_events.insert( {
   "expireAt": new Date('July 22, 2013 14:00:00'),
   "logEvent": 2,
   "logMessage": "Success!"
} )
```

MongoDB will automatically delete documents from the `log_events` collection when the documents' `expireAt` value is older than the number of seconds specified in `expireAfterSeconds`, i.e. `0` seconds older in this case. As such, the data expires at the specified `expireAt` value.

[81] If the field contains an array of BSON date-typed objects, data expires if at least one of BSON date-typed object is older than the number of seconds specified in `expireAfterSeconds`.

1.3 Optimization Strategies for MongoDB

There are many factors that can affect database performance and responsiveness including index use, query structure, data models and application design, as well as operational factors such as architecture and system configuration.

This section describes techniques for optimizing application performance with MongoDB.

***Analyzing MongoDB Performance* (page 37)** Discusses some of the factors that can influence MongoDB's performance.

***Evaluate Performance of Current Operations* (page 40)** MongoDB provides introspection tools that describe the query execution process, to allow users to test queries and build more efficient queries.

***Optimize Query Performance* (page 41)** Introduces the use of *projections* to reduce the amount of data MongoDB sends to clients.

***Design Notes* (page 43)** A collection of notes related to the architecture, design, and administration of MongoDB-based applications.

1.3.1 Analyzing MongoDB Performance

On this page

As you develop and operate applications with MongoDB, you may need to analyze the performance of the application and its database. When you encounter degraded performance, it is often a function of database access strategies, hardware availability, and the number of open database connections.

Some users may experience performance limitations as a result of inadequate or inappropriate indexing strategies, or as a consequence of poor schema design patterns. *Locking Performance* (page 37) discusses how these can impact MongoDB's internal locking.

Performance issues may indicate that the database is operating at capacity and that it is time to add additional capacity to the database. In particular, the application's *working set* should fit in the available physical memory. See *Memory and the MMAPv1 Storage Engine* (page 38) for more information on the working set.

In some cases performance issues may be temporary and related to abnormal traffic load. As discussed in *Number of Connections* (page 39), scaling can help relax excessive traffic.

Database Profiling (page 39) can help you to understand what operations are causing degradation.

Locking Performance

MongoDB uses a locking system to ensure data set consistency. If certain operations are long-running or a queue forms, performance will degrade as requests and operations wait for the lock.

Lock-related slowdowns can be intermittent. To see if the lock has been affecting your performance, refer to the *server-status-locks* section and the *globalLock* section of the `serverStatus` output.

Dividing `locks.timeAcquiringMicros` by `locks.acquireWaitCount` can give an approximate average wait time for a particular lock mode.

`locks.deadlockCount` provide the number of times the lock acquisitions encountered deadlocks.

If `globalLock.currentQueue.total` is consistently high, then there is a chance that a large number of requests are waiting for a lock. This indicates a possible concurrency issue that may be affecting performance.

If `globalLock.totalTime` is high relative to `uptime`, the database has existed in a lock state for a significant amount of time.

Long queries can result from ineffective use of indexes; non-optimal schema design; poor query structure; system architecture issues; or insufficient RAM resulting in *page faults* (page 38) and disk reads.

Memory and the MMAPv1 Storage Engine

Memory Use

With the *MMAPv1* storage engine, MongoDB uses memory-mapped files to store data. Given a data set of sufficient size, the `mongod` process will allocate all available memory on the system for its use.

While this is intentional and aids performance, the memory mapped files make it difficult to determine if the amount of RAM is sufficient for the data set.

The *memory usage statuses* metrics of the `serverStatus` output can provide insight into MongoDB's memory use.

The `mem.resident` field provides the amount of resident memory in use. If this exceeds the amount of system memory *and* there is a significant amount of data on disk that isn't in RAM, you may have exceeded the capacity of your system.

You can inspect `mem.mapped` to check the amount of mapped memory that `mongod` is using. If this value is greater than the amount of system memory, some operations will require a *page faults* to read data from disk.

Page Faults

With the MMAPv1 storage engine, page faults can occur as MongoDB reads from or writes data to parts of its data files that are not currently located in physical memory. In contrast, operating system page faults happen when physical memory is exhausted and pages of physical memory are swapped to disk.

MongoDB reports its triggered page faults as the total number of *page faults* in one second. To check for page faults, see the `extra_info.page_faults` value in the `serverStatus` output.

Rapid increases in the MongoDB page fault counter may indicate that the server has too little physical memory. Page faults also can occur while accessing large data sets or scanning an entire collection.

A single page fault completes quickly and is not problematic. However, in aggregate, large volumes of page faults typically indicate that MongoDB is reading too much data from disk.

MongoDB can often "yield" read locks after a page fault, allowing other database processes to read while `mongod` loads the next page into memory. Yielding the read lock following a page fault improves concurrency, and also improves overall throughput in high volume systems.

Increasing the amount of RAM accessible to MongoDB may help reduce the frequency of page faults. If this is not possible, you may want to consider deploying a *sharded cluster* or adding *shards* to your deployment to distribute load among `mongod` instances.

See *faq-storage-page-faults* for more information.

Number of Connections

In some cases, the number of connections between the applications and the database can overwhelm the ability of the server to handle requests. The following fields in the `serverStatus` document can provide insight:

- `globalLock.activeClients` contains a counter of the total number of clients with active operations in progress or queued.
- `connections` is a container for the following two fields:
  - `current` the total number of current clients that connect to the database instance.
  - `available` the total number of unused connections available for new clients.

If there are numerous concurrent application requests, the database may have trouble keeping up with demand. If this is the case, then you will need to increase the capacity of your deployment.

For read-heavy applications, increase the size of your *replica set* and distribute read operations to *secondary* members.

For write-heavy applications, deploy *sharding* and add one or more *shards* to a *sharded cluster* to distribute load among `mongod` instances.

Spikes in the number of connections can also be the result of application or driver errors. All of the officially supported MongoDB drivers implement connection pooling, which allows clients to use and reuse connections more efficiently. Extremely high numbers of connections, particularly without corresponding workload is often indicative of a driver or other configuration error.

Unless constrained by system-wide limits, MongoDB has no limit on incoming connections. On Unix-based systems, you can modify system limits using the `ulimit` command, or by editing your system's `/etc/sysctl` file. See *UNIX ulimit Settings* (page 125) for more information.

Database Profiling

MongoDB's "Profiler" is a database profiling system that can help identify inefficient queries and operations.

The following profiling levels are available:

Level	Setting
0	Off. No profiling
1	On. Only includes *"slow"* operations
2	On. Includes *all* operations

Enable the profiler by setting the `profile` value using the following command in the `mongo` shell:

```
db.setProfilingLevel(1)
```

The `slowOpThresholdMs` setting defines what constitutes a "slow" operation. To set the threshold above which the profiler considers operations "slow" (and thus, included in the level `1` profiling data), you can configure `slowOpThresholdMs` at runtime as an argument to the `db.setProfilingLevel()` operation.

See

The documentation of `db.setProfilingLevel()` for more information.

By default, `mongod` records all "slow" queries to its `log`, as defined by `slowOpThresholdMs`.

Note: Because the database profiler can negatively impact performance, only enable profiling for strategic intervals and as minimally as possible on production systems.

You may enable profiling on a per-`mongod` basis. This setting will not propagate across a *replica set* or *sharded cluster*.

You can view the output of the profiler in the `system.profile` collection of your database by issuing the `show profile` command in the `mongo` shell, or with the following operation:

```
db.system.profile.find( { millis : { $gt : 100 } } )
```

This returns all operations that lasted longer than 100 milliseconds. Ensure that the value specified here (`100`, in this example) is above the `slowOpThresholdMs` threshold.

You must use the `$query` operator to access the `query` field of documents within `system.profile`.

Additional Resources

- MongoDB Ops Optimization Consulting Package[82]

1.3.2 Evaluate Performance of Current Operations

On this page

The following sections describe techniques for evaluating operational performance.

Use the Database Profiler to Evaluate Operations Against the Database

MongoDB provides a database profiler that shows performance characteristics of each operation against the database. Use the profiler to locate any queries or write operations that are running slow. You can use this information, for example, to determine what indexes to create.

For more information, see *Database Profiling* (page 39).

Use `db.currentOp()` to Evaluate `mongod` Operations

The `db.currentOp()` method reports on current operations running on a `mongod` instance.

Use `explain` to Evaluate Query Performance

The `cursor.explain()` and `db.collection.explain()` methods return information on a query execution, such as the index MongoDB selected to fulfill the query and execution statistics. You can run the methods in *queryPlanner* mode, *executionStats* mode, or *allPlansExecution* mode to control the amount of information returned.

Example

[82] https://www.mongodb.com/products/consulting?jmp=docs#ops_optimization

To use `cursor.explain()` on a query for documents matching the expression `{ a: 1 }`, in the collection named `records`, use an operation that resembles the following in the `mongo` shell:

```
db.records.find( { a: 1 } ).explain("executionStats")
```

For more information, see `https://docs.mongodb.org/manual/reference/explain-results`, `cursor.explain()`, `db.collection.explain()`, and `https://docs.mongodb.org/manual/tutorial/an`

Additional Resources

- MongoDB Performance Evaluation and Tuning Consulting Package[83]

1.3.3 Optimize Query Performance

On this page

Create Indexes to Support Queries

For commonly issued queries, create `indexes`. If a query searches multiple fields, create a *compound index*. Scanning an index is much faster than scanning a collection. The indexes structures are smaller than the documents reference, and store references in order.

Example

If you have a `posts` collection containing blog posts, and if you regularly issue a query that sorts on the `author_name` field, then you can optimize the query by creating an index on the `author_name` field:

```
db.posts.createIndex( { author_name : 1 } )
```

Indexes also improve efficiency on queries that routinely sort on a given field.

Example

If you regularly issue a query that sorts on the `timestamp` field, then you can optimize the query by creating an index on the `timestamp` field:

Creating this index:

```
db.posts.createIndex( { timestamp : 1 } )
```

Optimizes this query:

[83] https://www.mongodb.com/products/consulting?jmp=docs#performance_evaluation

```
db.posts.find().sort( { timestamp : -1 } )
```

Because MongoDB can read indexes in both ascending and descending order, the direction of a single-key index does not matter.

Indexes support queries, update operations, and some phases of the *aggregation pipeline*.

Index keys that are of the `BinData` type are more efficiently stored in the index if:

- the binary subtype value is in the range of 0-7 or 128-135, and
- the length of the byte array is: 0, 1, 2, 3, 4, 5, 6, 7, 8, 10, 12, 14, 16, 20, 24, or 32.

Limit the Number of Query Results to Reduce Network Demand

MongoDB *cursors* return results in groups of multiple documents. If you know the number of results you want, you can reduce the demand on network resources by issuing the `limit()` method.

This is typically used in conjunction with sort operations. For example, if you need only 10 results from your query to the `posts` collection, you would issue the following command:

```
db.posts.find().sort( { timestamp : -1 } ).limit(10)
```

For more information on limiting results, see `limit()`

Use Projections to Return Only Necessary Data

When you need only a subset of fields from documents, you can achieve better performance by returning only the fields you need:

For example, if in your query to the `posts` collection, you need only the `timestamp`, `title`, `author`, and `abstract` fields, you would issue the following command:

```
db.posts.find( {}, { timestamp : 1 , title : 1 , author : 1 , abstract : 1} ).sort( { timestamp
```

For more information on using projections, see *read-operations-projection*.

Use `$hint` to Select a Particular Index

In most cases the *query optimizer* selects the optimal index for a specific operation; however, you can force MongoDB to use a specific index using the `hint()` method. Use `hint()` to support performance testing, or on some queries where you must select a field or field included in several indexes.

Use the Increment Operator to Perform Operations Server-Side

Use MongoDB's `$inc` operator to increment or decrement values in documents. The operator increments the value of the field on the server side, as an alternative to selecting a document, making simple modifications in the client and then writing the entire document to the server. The `$inc` operator can also help avoid race conditions, which would result when two application instances queried for a document, manually incremented a field, and saved the entire document back at the same time.

Additional Resources

- MongoDB Performance Evaluation and Tuning Consulting Package[84]

1.3.4 Design Notes

On this page

This page details features of MongoDB that may be important to keep in mind when developing applications.

Schema Considerations

Dynamic Schema

Data in MongoDB has a *dynamic schema*. *Collections* do not enforce *document* structure. This facilitates iterative development and polymorphism. Nevertheless, collections often hold documents with highly homogeneous structures. See `https://docs.mongodb.org/manual/core/data-models` for more information.

Some operational considerations include:

- the exact set of collections to be used;
- the indexes to be used: with the exception of the `_id` index, all indexes must be created explicitly;
- shard key declarations: choosing a good shard key is very important as the shard key cannot be changed once set.

Avoid importing unmodified data directly from a relational database. In general, you will want to "roll up" certain data into richer documents that take advantage of MongoDB's support for embedded documents and nested arrays.

Case Sensitive Strings

MongoDB strings are case sensitive. So a search for `"joe"` will not find `"Joe"`.

Consider:

- storing data in a normalized case format, or
- using regular expressions ending with the `i` option, and/or
- using `$toLower` or `$toUpper` in the `aggregation framework`.

[84]https://www.mongodb.com/products/consulting?jmp=docs#performance_evaluation

Type Sensitive Fields

MongoDB data is stored in the BSON format, a binary encoded serialization of JSON-like documents. BSON encodes additional type information. See bsonspec.org[85] for more information.

Consider the following document which has a field `x` with the *string* value `"123"`:

```
{ x : "123" }
```

Then the following query which looks for a *number* value `123` will **not** return that document:

```
db.mycollection.find( { x : 123 } )
```

General Considerations

By Default, Updates Affect one Document

To update multiple documents that meet your query criteria, set the `update multi` option to `true` or `1`. See: *Update Multiple Documents*.

Prior to MongoDB 2.2, you would specify the `upsert` and `multi` options in the `update` method as positional boolean options. See: the `update` method reference documentation.

BSON Document Size Limit

The `BSON Document Size` limit is currently set at 16MB per document. If you require larger documents, use `GridFS`.

No Fully Generalized Transactions

MongoDB does not have `fully generalized transactions`. If you model your data using rich documents that closely resemble your application's objects, each logical object will be in one MongoDB document. MongoDB allows you to modify a document in a single atomic operation. These kinds of data modification pattern covers most common uses of transactions in other systems.

Replica Set Considerations

Use an Odd Number of Replica Set Members

`Replica sets` perform consensus elections. To ensure that elections will proceed successfully, either use an odd number of members, typically three, or else use an *arbiter* to ensure an odd number of votes.

Keep Replica Set Members Up-to-Date

MongoDB replica sets support `automatic failover`. It is important for your secondaries to be up-to-date. There are various strategies for assessing consistency:

1. Use monitoring tools to alert you to lag events. See *Monitoring for MongoDB* (page 6) for a detailed discussion of MongoDB's monitoring options.

[85] http://bsonspec.org/#/specification

2. Specify appropriate write concern.
3. If your application requires *manual* fail over, you can configure your secondaries as *priority 0*. Priority 0 secondaries require manual action for a failover. This may be practical for a small replica set, but large deployments should fail over automatically.

See also:

replica set rollbacks.

Sharding Considerations

- Pick your shard keys carefully. You cannot choose a new shard key for a collection that is already sharded.
- Shard key values are immutable.
- When enabling sharding on an *existing collection*, MongoDB imposes a maximum size on those collections to ensure that it is possible to create chunks. For a detailed explanation of this limit, see: `<sharding-existing-collection-data-size>`.

  To shard large amounts of data, create a new empty sharded collection, and ingest the data from the source collection using an application level import operation.
- Unique indexes are not enforced across shards except for the shard key itself. See *Enforce Unique Keys for Sharded Collections* (page 256).
- Consider *pre-splitting* (page 246) an empty sharded collection before a massive bulk import.

Analyze Performance

As you develop and operate applications with MongoDB, you may want to analyze the performance of the database as the application. *Analyzing MongoDB Performance* (page 37) discusses some of the operational factors that can influence performance.

Additional Resources

- MongoDB Ops Optimization Consulting Package[86]

[86]https://www.mongodb.com/products/consulting?jmp=docs#ops_optimization

CHAPTER 2

Administration Tutorials

The administration tutorials provide specific step-by-step instructions for performing common MongoDB setup, maintenance, and configuration operations.

Configuration, Maintenance, and Analysis **(page 47)** Describes routine management operations, including configuration and performance analysis.

Manage mongod Processes **(page 52)** Start, configure, and manage running `mongod` process.

Rotate Log Files **(page 61)** Archive the current log files and start new ones.

Continue reading from *Configuration, Maintenance, and Analysis* (page 47) for additional tutorials of fundamental MongoDB maintenance procedures.

Backup and Recovery **(page 75)** Outlines procedures for data backup and restoration with `mongod` instances and deployments.

Backup and Restore with Filesystem Snapshots **(page 75)** An outline of procedures for creating MongoDB data set backups using system-level file snapshot tool, such as *LVM* or native storage appliance tools.

Backup and Restore Sharded Clusters **(page 88)** Detailed procedures and considerations for backing up sharded clusters and single shards.

Recover Data after an Unexpected Shutdown **(page 98)** Recover data from MongoDB data files that were not properly closed or have an invalid state.

Continue reading from *Backup and Recovery* (page 75) for additional tutorials of MongoDB backup and recovery procedures.

MongoDB Scripting **(page 101)** An introduction to the scripting capabilities of the `mongo` shell and the scripting capabilities embedded in MongoDB instances.

MongoDB Tutorials **(page 121)** A complete list of tutorials in the MongoDB Manual that address MongoDB operation and use.

2.1 Configuration, Maintenance, and Analysis

The following tutorials describe routine management operations, including configuration and performance analysis:

Disable Transparent Huge Pages (THP) **(page 48)** Describes Transparent Huge Pages (THP) and provides detailed instructions on disabling them.

***Use Database Commands* (page 51)** The process for running database commands that provide basic database operations.

***Manage mongod Processes* (page 52)** Start, configure, and manage running `mongod` process.

***Terminate Running Operations* (page 55)** Stop in progress MongoDB client operations using `db.killOp()` and `maxTimeMS()`.

***Analyze Performance of Database Operations* (page 56)** Collect data that introspects the performance of query and update operations on a `mongod` instance.

***Rotate Log Files* (page 61)** Archive the current log files and start new ones.

***Manage Journaling* (page 63)** Describes the procedures for configuring and managing MongoDB's journaling system which allows MongoDB to provide crash resiliency and durability.

***Store a JavaScript Function on the Server* (page 65)** Describes how to store JavaScript functions on a MongoDB server.

***Upgrade to the Latest Revision of MongoDB* (page 66)** Introduces the basic process for upgrading a MongoDB deployment between different minor release versions.

***Monitor MongoDB With SNMP on Linux* (page 69)** The SNMP extension, available in MongoDB Enterprise, allows MongoDB to provide database metrics via SNMP.

***Monitor MongoDB Windows with SNMP* (page 71)** The SNMP extension, available in the Windows build of MongoDB Enterprise, allows MongoDB to provide database metrics via SNMP.

***Troubleshoot SNMP* (page 73)** Outlines common errors and diagnostic processes useful for deploying MongoDB Enterprise with SNMP support.

2.1.1 Disable Transparent Huge Pages (THP)

On this page

Transparent Huge Pages (THP) is a Linux memory management system that reduces the overhead of Translation Lookaside Buffer (TLB) lookups on machines with large amounts of memory by using larger memory pages.

However, database workloads often perform poorly with THP, because they tend to have sparse rather than contiguous memory access patterns. You should disable THP on Linux machines to ensure best performance with MongoDB.

Init Script

Important: If you are using `tuned` or `ktune` (for example, if you are running Red Hat or CentOS 6+), you must additionally configure them so that THP is not re-enabled. See *Using tuned and ktune* (page 50).

Step 1: Create the `init.d` script.

Create the following file at `/etc/init.d/disable-transparent-hugepages`:

```
#!/bin/sh
### BEGIN INIT INFO
# Provides:          disable-transparent-hugepages
# Required-Start:    $local_fs
# Required-Stop:
# X-Start-Before:    mongod mongodb-mms-automation-agent
# Default-Start:     2 3 4 5
# Default-Stop:      0 1 6
# Short-Description: Disable Linux transparent huge pages
# Description:       Disable Linux transparent huge pages, to improve
#                    database performance.
### END INIT INFO

case $1 in
  start)
    if [ -d /sys/kernel/mm/transparent_hugepage ]; then
      thp_path=/sys/kernel/mm/transparent_hugepage
    elif [ -d /sys/kernel/mm/redhat_transparent_hugepage ]; then
      thp_path=/sys/kernel/mm/redhat_transparent_hugepage
    else
      return 0
    fi

    echo 'never' > ${thp_path}/enabled
    echo 'never' > ${thp_path}/defrag

    unset thp_path
    ;;
esac
```

Step 2: Make it executable.

Run the following command to ensure that the init script can be used:

```
sudo chmod 755 /etc/init.d/disable-transparent-hugepages
```

Step 3: Configure your operating system to run it on boot.

Use the appropriate command to configure the new init script on your Linux distribution.

Distribution	Command
Ubuntu and Debian	`sudo update-rc.d disable-transparent-hugepages d`
SUSE	`sudo insserv /etc/init.d/disable-transparent-hug`
Red Hat, CentOS, Amazon Linux, and derivatives	`sudo chkconfig --add disable-transparent-hugepag`

Step 4: Override tuned and ktune, if applicable

If you are using `tuned` or `ktune` (for example, if you are running Red Hat or CentOS 6+) you must now configure them to preserve the above settings.

Using `tuned` and `ktune`

Important: If using `tuned` or `ktune`, you must perform this step in addition to installing the init script.

`tuned` and `ktune` are dynamic kernel tuning tools available on Red Hat and CentOS that can disable transparent huge pages.

To disable transparent huge pages in `tuned` or `ktune`, you need to edit or create a new profile that sets THP to `never`.

Red Hat/CentOS 6

Step 1: Create a new profile. Create a new profile from an existing default profile by copying the relevant directory. In the example we use the `default` profile as the base and call our new profile `no-thp`.

```
sudo cp -r /etc/tune-profiles/default /etc/tune-profiles/no-thp
```

Step 2: Edit `ktune.sh`. Edit `/etc/tune-profiles/no-thp/ktune.sh` and add the following:

```
set_transparent_hugepages never
```

to the `start()` block of the file, before the `return 0` statement.

Step 3: Enable the new profile. Finally, enable the new profile by issuing:

```
sudo tuned-adm profile no-thp
```

Red Hat/CentOS 7

Step 1: Create a new profile. Create a new `tuned` profile directory:

```
sudo mkdir /etc/tuned/no-thp
```

Step 2: Edit `tuned.conf`. Create and edit `/etc/tuned/no-thp/tuned.conf` so that it contains the following:

```
[main]
include=virtual-guest

[vm]
transparent_hugepages=never
```

Step 3: Enable the new profile. Finally, enable the new profile by issuing:

```
sudo tuned-adm profile no-thp
```

Test Your Changes

You can check the status of THP support by issuing the following commands:

```
cat /sys/kernel/mm/transparent_hugepage/enabled
cat /sys/kernel/mm/transparent_hugepage/defrag
```

On Red Hat Enterprise Linux, CentOS, and potentially other Red Hat-based derivatives, you may instead need to use the following:

```
cat /sys/kernel/mm/redhat_transparent_hugepage/enabled
cat /sys/kernel/mm/redhat_transparent_hugepage/defrag
```

For both files, the correct output resembles:

```
always madvise [never]
```

2.1.2 Use Database Commands

On this page

The MongoDB command interface provides access to all *non CRUD* database operations. Fetching server stats, initializing a replica set, and running a map-reduce job are all accomplished with commands.

See `https://docs.mongodb.org/manual/reference/command` for list of all commands sorted by function.

Database Command Form

You specify a command first by constructing a standard *BSON* document whose first key is the name of the command. For example, specify the `isMaster` command using the following *BSON* document:

```
{ isMaster: 1 }
```

Issue Commands

The `mongo` shell provides a helper method for running commands called `db.runCommand()`. The following operation in `mongo` runs the above command:

```
db.runCommand( { isMaster: 1 } )
```

Many drivers provide an equivalent for the db.runCommand() method. Internally, running commands with db.runCommand() is equivalent to a special query against the *$cmd* collection.

Many common commands have their own shell helpers or wrappers in the mongo shell and drivers, such as the db.isMaster() method in the mongo JavaScript shell.

You can use the maxTimeMS option to specify a time limit for the execution of a command, see *Terminate a Command* (page 56) for more information on operation termination.

admin Database Commands

You must run some commands on the *admin database*. Normally, these operations resemble the followings:

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

However, there's also a command helper that automatically runs the command in the context of the admin database:

```
db._adminCommand( {buildInfo: 1} )
```

Command Responses

All commands return, at minimum, a document with an ok field indicating whether the command has succeeded:

```
{ 'ok': 1 }
```

Failed commands return the ok field with a value of 0.

2.1.3 Manage mongod Processes

On this page

MongoDB runs as a standard program. You can start MongoDB from a command line by issuing the mongod command and specifying options. For a list of options, see the mongod reference. MongoDB can also run as a Windows service. For details, see *manually-create-windows-service*. To install MongoDB, see https://docs.mongodb.org/manual/installation.

The following examples assume the directory containing the mongod process is in your system paths. The mongod process is the primary database process that runs on an individual server. mongos provides a coherent MongoDB interface equivalent to a mongod from the perspective of a client. The mongo binary provides the administrative shell.

This document discusses the mongod process; however, some portions of this document may be applicable to mongos instances.

Start `mongod` Processes

By default, MongoDB stores data in the `/data/db` directory. On Windows, MongoDB stores data in `C:\data\db`. On all platforms, MongoDB listens for connections from clients on port `27017`.

To start MongoDB using all defaults, issue the following command at the system shell:

```
mongod
```

Specify a Data Directory

If you want `mongod` to store data files at a path *other than* `/data/db` you can specify a `dbPath`. The `dbPath` must exist before you start `mongod`. If it does not exist, create the directory and the permissions so that `mongod` can read and write data to this path. For more information on permissions, see the `security operations documentation`.

To specify a `dbPath` for `mongod` to use as a data directory, use the *`--dbpath`* option. The following invocation will start a `mongod` instance and store data in the `/srv/mongodb` path

```
mongod --dbpath /srv/mongodb/
```

Specify a TCP Port

Only a single process can listen for connections on a network interface at a time. If you run multiple `mongod` processes on a single machine, or have other processes that must use this port, you must assign each a different port to listen on for client connections.

To specify a port to `mongod`, use the *`--port`* option on the command line. The following command starts `mongod` listening on port `12345`:

```
mongod --port 12345
```

Use the default port number when possible, to avoid confusion.

Start `mongod` as a Daemon

To run a `mongod` process as a daemon (i.e. `fork`), *and* write its output to a log file, use the *`--fork`* and *`--logpath`* options. You must create the log directory; however, `mongod` will create the log file if it does not exist.

The following command starts `mongod` as a daemon and records log output to `/var/log/mongodb.log`.

```
mongod --fork --logpath /var/log/mongodb.log
```

Additional Configuration Options

For an overview of common configurations and deployments for common use cases, see *Run-time Database Configuration* (page 12).

Stop `mongod` Processes

In a clean shutdown a `mongod` completes all pending operations, flushes all data to data files, and closes all data files. Other shutdowns are *unclean* and can compromise the validity of the data files.

To ensure a clean shutdown, always shutdown `mongod` instances using one of the following methods:

Use `shutdownServer()`

Shut down the `mongod` from the `mongo` shell using the `db.shutdownServer()` method as follows:

```
use admin
db.shutdownServer()
```

Calling the same method from a *init script* accomplishes the same result.

For systems with `authorization` enabled, users may only issue `db.shutdownServer()` when authenticated to the `admin` database or via the localhost interface on systems without authentication enabled.

Use `--shutdown`

From the Linux command line, shut down the `mongod` using the `--shutdown` option in the following command:

```
mongod --shutdown
```

Use `CTRL-C`

When running the `mongod` instance in interactive mode (i.e. without `--fork`), issue `Control-C` to perform a clean shutdown.

Use `kill`

From the Linux command line, shut down a specific `mongod` instance using the following command:

```
kill <mongod process ID>
```

Warning: Never use `kill -9` (i.e. `SIGKILL`) to terminate a mongod instance.

Stop a Replica Set

Procedure

If the `mongod` is the *primary* in a *replica set*, the shutdown process for this `mongod` instance has the following steps:

1. Check how up-to-date the *secondaries* are.
2. If no secondary is within 10 seconds of the primary, `mongod` will return a message that it will not shut down. You can pass the `shutdown` command a `timeoutSecs` argument to wait for a secondary to catch up.

3. If there is a secondary within 10 seconds of the primary, the primary will step down and wait for the secondary to catch up.
4. After 60 seconds or once the secondary has caught up, the primary will shut down.

Force Replica Set Shutdown

If there is no up-to-date secondary and you want the primary to shut down, issue the `shutdown` command with the `force` argument, as in the following `mongo` shell operation:

```
db.adminCommand({shutdown : 1, force : true})
```

To keep checking the secondaries for a specified number of seconds if none are immediately up-to-date, issue `shutdown` with the `timeoutSecs` argument. MongoDB will keep checking the secondaries for the specified number of seconds if none are immediately up-to-date. If any of the secondaries catch up within the allotted time, the primary will shut down. If no secondaries catch up, it will not shut down.

The following command issues `shutdown` with `timeoutSecs` set to `5`:

```
db.adminCommand({shutdown : 1, timeoutSecs : 5})
```

Alternately you can use the `timeoutSecs` argument with the `db.shutdownServer()` method:

```
db.shutdownServer({timeoutSecs : 5})
```

2.1.4 Terminate Running Operations

On this page

Overview

MongoDB provides two facilitates to terminate running operations: `maxTimeMS()` and `db.killOp()`. Use these operations as needed to control the behavior of operations in a MongoDB deployment.

Available Procedures

`maxTimeMS`

New in version 2.6.

The `maxTimeMS()` method sets a time limit for an operation. When the operation reaches the specified time limit, MongoDB interrupts the operation at the next *interrupt point*.

Terminate a Query From the `mongo` shell, use the following method to set a time limit of 30 milliseconds for this query:

```
db.location.find( { "town": { "$regex": "(Pine Lumber)",
                              "$options": 'i' } } ).maxTimeMS(30)
```

Terminate a Command Consider a potentially long running operation using `distinct` to return each distinct``collection`` field that has a `city` key:

```
db.runCommand( { distinct: "collection",
                 key: "city" } )
```

You can add the `maxTimeMS` field to the command document to set a time limit of 45 milliseconds for the operation:

```
db.runCommand( { distinct: "collection",
                 key: "city",
                 maxTimeMS: 45 } )
```

`db.getLastError()` and `db.getLastErrorObj()` will return errors for interrupted options:

```
{ "n" : 0,
  "connectionId" : 1,
  "err" : "operation exceeded time limit",
  "ok" : 1 }
```

`killOp`

The `db.killOp()` method interrupts a running operation at the next *interrupt point*. `db.killOp()` identifies the target operation by operation ID.

```
db.killOp(<opId>)
```

Warning: Terminate running operations with extreme caution. Only use `db.killOp()` to terminate operations initiated by clients and *do not* terminate internal database operations.

Related

To return a list of running operations see `db.currentOp()`.

2.1.5 Analyze Performance of Database Operations

On this page

The database profiler collects fine grained data about MongoDB write operations, cursors, database commands on a running `mongod` instance. You can enable profiling on a per-database or per-instance basis. The *profiling level* (page 57) is also configurable when enabling profiling.

The database profiler writes all the data it collects to the `system.profile` (page 130) collection, which is a *capped collection* (page 32). See *Database Profiler Output* (page 130) for overview of the data in the `system.profile` (page 130) documents created by the profiler.

This document outlines a number of key administration options for the database profiler. For additional related information, consider the following resources:

- *Database Profiler Output* (page 130)
- `Profile Command`
- `db.currentOp()`

Profiling Levels

The following profiling levels are available:

- `0` - the profiler is off, does not collect any data. `mongod` always writes operations longer than the `slowOpThresholdMs` threshold to its log.
- `1` - collects profiling data for slow operations only. By default slow operations are those slower than 100 milliseconds.

  You can modify the threshold for "slow" operations with the `slowOpThresholdMs` runtime option or the `setParameter` command. See the *Specify the Threshold for Slow Operations* (page 57) section for more information.
- `2` - collects profiling data for all database operations.

Enable Database Profiling and Set the Profiling Level

You can enable database profiling from the `mongo` shell or through a driver using the `profile` command. This section will describe how to do so from the `mongo` shell. See your `driver documentation` if you want to control the profiler from within your application.

When you enable profiling, you also set the *profiling level* (page 57). The profiler records data in the `system.profile` (page 130) collection. MongoDB creates the `system.profile` (page 130) collection in a database after you enable profiling for that database.

To enable profiling and set the profiling level, use the `db.setProfilingLevel()` helper in the `mongo` shell, passing the profiling level as a parameter. For example, to enable profiling for all database operations, consider the following operation in the `mongo` shell:

```
db.setProfilingLevel(2)
```

The shell returns a document showing the *previous* level of profiling. The `"ok" : 1` key-value pair indicates the operation succeeded:

```
{ "was" : 0, "slowms" : 100, "ok" : 1 }
```

To verify the new setting, see the *Check Profiling Level* (page 58) section.

Specify the Threshold for Slow Operations

The threshold for slow operations applies to the entire `mongod` instance. When you change the threshold, you change it for all databases on the instance.

Important: Changing the slow operation threshold for the database profiler also affects the profiling subsystem's slow operation threshold for the entire `mongod` instance. Always set the threshold to the highest useful value.

By default the slow operation threshold is 100 milliseconds. Databases with a profiling level of 1 will log operations slower than 100 milliseconds.

To change the threshold, pass two parameters to the db.setProfilingLevel() helper in the mongo shell. The first parameter sets the profiling level for the current database, and the second sets the default slow operation threshold *for the entire* mongod *instance*.

For example, the following command sets the profiling level for the current database to 0, which disables profiling, and sets the slow-operation threshold for the mongod instance to 20 milliseconds. Any database on the instance with a profiling level of 1 will use this threshold:

```
db.setProfilingLevel(0,20)
```

Check Profiling Level

To view the *profiling level* (page 57), issue the following from the mongo shell:

```
db.getProfilingStatus()
```

The shell returns a document similar to the following:

```
{ "was" : 0, "slowms" : 100 }
```

The was field indicates the current level of profiling.

The slowms field indicates how long an operation must exist in milliseconds for an operation to pass the "slow" threshold. MongoDB will log operations that take longer than the threshold if the profiling level is 1. This document returns the profiling level in the was field. For an explanation of profiling levels, see *Profiling Levels* (page 57).

To return only the profiling level, use the db.getProfilingLevel() helper in the mongo as in the following:

```
db.getProfilingLevel()
```

Disable Profiling

To disable profiling, use the following helper in the mongo shell:

```
db.setProfilingLevel(0)
```

Enable Profiling for an Entire mongod Instance

For development purposes in testing environments, you can enable database profiling for an entire mongod instance. The profiling level applies to all databases provided by the mongod instance.

To enable profiling for a mongod instance, pass the following parameters to mongod at startup or within the configuration file:

```
mongod --profile=1 --slowms=15
```

This sets the profiling level to 1, which collects profiling data for slow operations only, and defines slow operations as those that last longer than 15 milliseconds.

See also:

mode and slowOpThresholdMs.

Database Profiling and Sharding

You *cannot* enable profiling on a `mongos` instance. To enable profiling in a shard cluster, you must enable profiling for each `mongod` instance in the cluster.

View Profiler Data

The database profiler logs information about database operations in the `system.profile` (page 130) collection.

To view profiling information, query the `system.profile` (page 130) collection. You can use `$comment` to add data to the query document to make it easier to analyze data from the profiler. To view example queries, see *Profiler Overhead* (page 60).

For an explanation of the output data, see *Database Profiler Output* (page 130).

Example Profiler Data Queries

This section displays example queries to the `system.profile` (page 130) collection. For an explanation of the query output, see *Database Profiler Output* (page 130).

To return the most recent 10 log entries in the `system.profile` (page 130) collection, run a query similar to the following:

```
db.system.profile.find().limit(10).sort( { ts : -1 } ).pretty()
```

To return all operations except command operations (*$cmd*), run a query similar to the following:

```
db.system.profile.find( { op: { $ne : 'command' } } ).pretty()
```

To return operations for a particular collection, run a query similar to the following. This example returns operations in the `mydb` database's `test` collection:

```
db.system.profile.find( { ns : 'mydb.test' } ).pretty()
```

To return operations slower than `5` milliseconds, run a query similar to the following:

```
db.system.profile.find( { millis : { $gt : 5 } } ).pretty()
```

To return information from a certain time range, run a query similar to the following:

```
db.system.profile.find(
                       {
                        ts : {
                              $gt : new ISODate("2012-12-09T03:00:00Z") ,
                              $lt : new ISODate("2012-12-09T03:40:00Z")
                             }
                       }
                      ).pretty()
```

The following example looks at the time range, suppresses the `user` field from the output to make it easier to read, and sorts the results by how long each operation took to run:

```
db.system.profile.find(
                       {
                         ts : {
                               $gt : new ISODate("2011-07-12T03:00:00Z") ,
                               $lt : new ISODate("2011-07-12T03:40:00Z")
```

```
                  }
              },
              { user : 0 }
            ).sort( { millis : -1 } )
```

Show the Five Most Recent Events

On a database that has profiling enabled, the `show profile` helper in the `mongo` shell displays the 5 most recent operations that took at least 1 millisecond to execute. Issue `show profile` from the `mongo` shell, as follows:

```
show profile
```

Profiler Overhead

When enabled, profiling has a minor effect on performance. The `system.profile` (page 130) collection is a *capped collection* with a default size of 1 megabyte. A collection of this size can typically store several thousand profile documents, but some application may use more or less profiling data per operation.

Change Size of `system.profile` Collection on the Primary

To change the size of the `system.profile` (page 130) collection, you must:

1. Disable profiling.
2. Drop the `system.profile` (page 130) collection.
3. Create a new `system.profile` (page 130) collection.
4. Re-enable profiling.

For example, to create a new `system.profile` (page 130) collections that's `4000000` bytes, use the following sequence of operations in the `mongo` shell:

```
db.setProfilingLevel(0)

db.system.profile.drop()

db.createCollection( "system.profile", { capped: true, size:4000000 } )

db.setProfilingLevel(1)
```

Change Size of `system.profile` Collection on a Secondary

To change the size of the `system.profile` (page 130) collection on a *secondary*, you must stop the secondary, run it as a standalone, and then perform the steps above. When done, restart the standalone as a member of the replica set. For more information, see *Perform Maintenance on Replica Set Members* (page 190).

Additional Resources

- MongoDB Performance Evaluation and Tuning Consulting Package[1]

[1] https://www.mongodb.com/products/consulting?jmp=docs#performance_evaluation

2.1.6 Rotate Log Files

On this page

Overview

When used with the `--logpath` option or `systemLog.path` setting, `mongod` and `mongos` instances report a live account of all activity and operations to a log file. When reporting activity data to a log file, by default, MongoDB only rotates logs in response to the `logRotate` command, or when the `mongod` or `mongos` process receives a `SIGUSR1` signal from the operating system.

MongoDB's standard log rotation approach archives the current log file and starts a new one. To do this, the `mongod` or `mongos` instance renames the current log file by appending a UTC timestamp to the filename, in *ISODate* format. It then opens a new log file, closes the old log file, and sends all new log entries to the new log file.

You can also configure MongoDB to support the Linux/Unix logrotate utility by setting `systemLog.logRotate` or `--logRotate` to `reopen`. With `reopen`, `mongod` or `mongos` closes the log file, and then reopens a log file with the same name, expecting that another process renamed the file prior to rotation.

Finally, you can configure `mongod` to send log data to the `syslog`. using the `--syslog` option. In this case, you can take advantage of alternate logrotation tools.

See also:

For information on logging, see the *Process Logging* (page 10) section.

Default Log Rotation Behavior

By default, MongoDB uses the `--logRotate rename` behavior. With `rename`, `mongod` or `mongos` renames the current log file by appending a UTC timestamp to the filename, opens a new log file, closes the old log file, and sends all new log entries to the new log file.

Step 1: Start a `mongod` instance.

```
mongod -v --logpath /var/log/mongodb/server1.log
```

You can also explicitly specify `logRotate --rename`.

Step 2: List the log files

In a separate terminal, list the matching files:

```
ls /var/log/mongodb/server1.log*
```

The results should include one log file, `server1.log`.

Step 3: Rotate the log file.

Rotate the log file by issuing the `logRotate` command from the `admin` database in a `mongo` shell:

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

Step 4: View the new log files

List the new log files to view the newly-created log:

```
ls /var/log/mongodb/server1.log*
```

There should be two log files listed: `server1.log`, which is the log file that `mongod` or `mongos` made when it reopened the log file, and `server1.log.<timestamp>`, the renamed original log file.

Rotating log files does not modify the "old" rotated log files. When you rotate a log, you rename the `server1.log` file to include the timestamp, and a new, empty `server1.log` file receives all new log input.

Log Rotation with `--logRotate reopen`

New in version 3.0.0.

Log rotation with *`--logRotate reopen`* closes and opens the log file following the typical Linux/Unix log rotate behavior.

Step 1: Start a `mongod` instance, specifying the `reopen` *`--logRotate`* behavior.

```
mongod -v --logpath /var/log/mongodb/server1.log --logRotate reopen --logappend
```

You must use the *`--logappend`* option with *`--logRotate reopen`*.

Step 2: List the log files

In a separate terminal, list the matching files:

```
ls /var/log/mongodb/server1.log*
```

The results should include one log file, `server1.log`.

Step 3: Rotate the log file.

Rotate the log file by issuing the `logRotate` command from the `admin` database in a `mongo` shell:

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

You should rename the log file using an external process, following the typical Linux/Unix log rotate behavior.

Syslog Log Rotation

New in version 2.2.

With syslog log rotation, `mongod` sends log data to the syslog rather than writing it to a file.

Step 1: Start a `mongod` instance with the `--syslog` option

```
mongod --syslog
```

Do not include *`--logpath`*. Since *`--syslog`* tells `mongod` to send log data to the syslog, specifying a *`--logpath`* will causes an error.

To specify the facility level used when logging messages to the syslog, use the *`--syslogFacility`* option or `systemLog.syslogFacility` configuration setting.

Step 2: Rotate the log.

Store and rotate the log output using your systems default log rotation mechanism.

Forcing a Log Rotation with `SIGUSR1`

For Linux and Unix-based systems, you can use the `SIGUSR1` signal to rotate the logs for a single process, as in the following:

```
kill -SIGUSR1 <mongod process id>
```

2.1.7 Manage Journaling

On this page

MongoDB uses *write ahead logging* to an on-disk *journal* to guarantee `write operation` durability and to provide crash resiliency.

With MMAPv1, before applying a change to the data files, MongoDB writes the change operation to the journal. If MongoDB should terminate or encounter an error before it can write the changes from the journal to the data files, MongoDB can re-apply the write operation and maintain a consistent state. By default, the greatest extent of lost writes, i.e., those not made to the journal, are those made in the last 100 milliseconds. See `commitIntervalMs` for more information on the default.

With WiredTiger, even without journaling, *checkpoints* provide a consistent view of data on disk and allow MongoDB to recover from the last checkpoint. However, if MongoDB exits unexpectedly in between checkpoints, journaling is required to recover information that occured after the last checkpoint.

Procedures

Enable Journaling

Changed in version 2.0: For 64-bit builds of `mongod`, journaling is enabled by default.

To enable journaling, start `mongod` with the `--journal` command line option.

Disable Journaling

Warning: Do not disable journaling on production systems. If your `mongod` instance stops without shutting down cleanly unexpectedly for any reason, (e.g. power failure) and you are not running with journaling, then you must recover from an unaffected *replica set* member or backup, as described in *repair* (page 98).

To disable journaling, start `mongod` with the `--nojournal` command line option.

Get Commit Acknowledgment

You can get commit acknowledgment with the *write-concern* and the `j` option. For details, see *write-concern-operation*.

Avoid Preallocation Lag for MMAPv1

With the `MMAPv1 storage engine`, MongoDB may preallocate journal files if the `mongod` process determines that it is more efficient to preallocate journal files than create new journal files as needed.

Depending on your filesystem, you might experience a preallocation lag the first time you start a `mongod` instance with journaling enabled. The amount of time required to pre-allocate files might last several minutes; during this time, you will not be able to connect to the database. This is a one-time preallocation and does not occur with future invocations.

To avoid *preallocation lag* (page 149), you can preallocate files in the journal directory by copying them from another instance of `mongod`.

Preallocated files do not contain data. It is safe to later remove them. But if you restart `mongod` with journaling, `mongod` will create them again.

Example

The following sequence preallocates journal files for an instance of `mongod` running on port `27017` with a database path of `/data/db`.

For demonstration purposes, the sequence starts by creating a set of journal files in the usual way.

1. Create a temporary directory into which to create a set of journal files:

```
mkdir ~/tmpDbpath
```

2. Create a set of journal files by staring a `mongod` instance that uses the temporary directory:

```
mongod --port 10000 --dbpath ~/tmpDbpath --journal
```

3. When you see the following log output, indicating `mongod` has the files, press CONTROL+C to stop the `mongod` instance:

```
[initandlisten] waiting for connections on port 10000
```

4. Preallocate journal files for the new instance of `mongod` by moving the journal files from the data directory of the existing instance to the data directory of the new instance:

```
mv ~/tmpDbpath/journal /data/db/
```

5. Start the new `mongod` instance:

```
mongod --port 27017 --dbpath /data/db --journal
```

Monitor Journal Status

Use the following commands and methods to monitor journal status:

- `serverStatus`

  The `serverStatus` command returns database status information that is useful for assessing performance.

- `journalLatencyTest`

  Use `journalLatencyTest` to measure how long it takes on your volume to write to the disk in an append-only fashion. You can run this command on an idle system to get a baseline sync time for journaling. You can also run this command on a busy system to see the sync time on a busy system, which may be higher if the journal directory is on the same volume as the data files.

  The `journalLatencyTest` command also provides a way to check if your disk drive is buffering writes in its local cache. If the number is very low (i.e., less than 2 milliseconds) and the drive is non-SSD, the drive is probably buffering writes. In that case, enable cache write-through for the device in your operating system, unless you have a disk controller card with battery backed RAM.

Change the Group Commit Interval for MMAPv1

For the `MMAPv1 storage engine`, you can set the group commit interval using the *`--journalCommitInterval`* command line option. The allowed range is `2` to `300` milliseconds.

Lower values increase the durability of the journal at the expense of disk performance.

Recover Data After Unexpected Shutdown

On a restart after a crash, MongoDB replays all journal files in the journal directory before the server becomes available. If MongoDB must replay journal files, `mongod` notes these events in the log output.

There is no reason to run `repairDatabase` in these situations.

2.1.8 Store a JavaScript Function on the Server

Note: Do not store application logic in the database. There are performance limitations to running JavaScript inside of MongoDB. Application code also is typically most effective when it shares version control with the application itself.

There is a special system collection named `system.js` that can store JavaScript functions for reuse.

To store a function, you can use the `db.collection.save()`, as in the following examples:

```
db.system.js.save(
   {
     _id: "echoFunction",
     value : function(x) { return x; }
   }
)

db.system.js.save(
   {
     _id : "myAddFunction" ,
     value : function (x, y){ return x + y; }
   }
);
```

- The `_id` field holds the name of the function and is unique per database.
- The `value` field holds the function definition.

Once you save a function in the `system.js` collection, you can use the function from any JavaScript context; e.g. `$where` operator, `mapReduce` command or `db.collection.mapReduce()`.

In the `mongo` shell, you can use `db.loadServerScripts()` to load all the scripts saved in the `system.js` collection for the current database. Once loaded, you can invoke the functions directly in the shell, as in the following example:

```
db.loadServerScripts();

echoFunction(3);

myAddFunction(3, 5);
```

2.1.9 Upgrade to the Latest Revision of MongoDB

On this page

Revisions provide security patches, bug fixes, and new or changed features that do not contain any backward breaking changes. Always upgrade to the latest revision in your release series. The third number in the *MongoDB version number* indicates the revision.

Before Upgrading

- Ensure you have an up-to-date backup of your data set. See *MongoDB Backup Methods* (page 4).

- Consult the following documents for any special considerations or compatibility issues specific to your MongoDB release:
  - The release notes, located at `https://docs.mongodb.org/manual/release-notes`.
  - The documentation for your driver. See Drivers[2] and Driver Compatibility[3] pages for more information.
- If your installation includes *replica sets*, plan the upgrade during a predefined maintenance window.
- Before you upgrade a production environment, use the procedures in this document to upgrade a *staging* environment that reproduces your production environment, to ensure that your production configuration is compatible with all changes.

Upgrade Procedure

Important: Always backup all of your data before upgrading MongoDB.

Upgrade each `mongod` and `mongos` binary separately, using the procedure described here. When upgrading a binary, use the procedure *Upgrade a MongoDB Instance* (page 67).

Follow this upgrade procedure:

1. For deployments that use authentication, first upgrade all of your MongoDB `drivers`. To upgrade, see the documentation for your driver as well as the Driver Compatibility[4] page.
2. Upgrade sharded clusters, as described in *Upgrade Sharded Clusters* (page 68).
3. Upgrade any standalone instances. See *Upgrade a MongoDB Instance* (page 67).
4. Upgrade any replica sets that are not part of a sharded cluster, as described in *Upgrade Replica Sets* (page 68).

Upgrade a MongoDB Instance

To upgrade a `mongod` or `mongos` instance, use one of the following approaches:

- Upgrade the instance using the operating system's package management tool and the official MongoDB packages. This is the preferred approach. See `https://docs.mongodb.org/manual/installation`.
- Upgrade the instance by replacing the existing binaries with new binaries. See *Replace the Existing Binaries* (page 67).

Replace the Existing Binaries

Important: Always backup all of your data before upgrading MongoDB.

This section describes how to upgrade MongoDB by replacing the existing binaries. The preferred approach to an upgrade is to use the operating system's package management tool and the official MongoDB packages, as described in `https://docs.mongodb.org/manual/installation`.

To upgrade a `mongod` or `mongos` instance by replacing the existing binaries:

[2] https://docs.mongodb.org/ecosystem/drivers
[3] https://docs.mongodb.org/ecosystem/drivers/driver-compatibility-reference
[4] https://docs.mongodb.org/ecosystem/drivers/driver-compatibility-reference

1. Download the binaries for the latest MongoDB revision from the MongoDB Download Page[5] and store the binaries in a temporary location. The binaries download as compressed files that uncompress to the directory structure used by the MongoDB installation.
2. Shutdown the instance.
3. Replace the existing MongoDB binaries with the downloaded binaries.
4. Restart the instance.

Upgrade Sharded Clusters

To upgrade a sharded cluster:

1. Disable the cluster's balancer, as described in *Disable the Balancer* (page 240).
2. Upgrade each `mongos` instance by following the instructions below in *Upgrade a MongoDB Instance* (page 67). You can upgrade the `mongos` instances in any order.
3. Upgrade each `mongod` *config server* individually starting with the last config server listed in your *`mongos --configdb`* string and working backward. To keep the cluster online, make sure at least one config server is always running. For each config server upgrade, follow the instructions below in *Upgrade a MongoDB Instance* (page 67)

   Example

   Given the following config string:

   ```
   mongos --configdb cfg0.example.net:27019,cfg1.example.net:27019,cfg2.example.net:27019
   ```

   You would upgrade the config servers in the following order:

   (a) cfg2.example.net

   (b) cfg1.example.net

   (c) cfg0.example.net

4. Upgrade each shard.
   - If a shard is a replica set, upgrade the shard using the procedure below titled *Upgrade Replica Sets* (page 68).
   - If a shard is a standalone instance, upgrade the shard using the procedure below titled *Upgrade a MongoDB Instance* (page 67).
5. Re-enable the balancer, as described in *Enable the Balancer* (page 241).

Upgrade Replica Sets

To upgrade a replica set, upgrade each member individually, starting with the *secondaries* and finishing with the *primary*. Plan the upgrade during a predefined maintenance window.

Upgrade Secondaries

Upgrade each secondary separately as follows:

[5] http://downloads.mongodb.org/

1. Upgrade the secondary's `mongod` binary by following the instructions below in *Upgrade a MongoDB Instance* (page 67).
2. After upgrading a secondary, wait for the secondary to recover to the `SECONDARY` state before upgrading the next instance. To check the member's state, issue `rs.status()` in the `mongo` shell.

   The secondary may briefly go into `STARTUP2` or `RECOVERING`. This is normal. Make sure to wait for the secondary to fully recover to `SECONDARY` before you continue the upgrade.

Upgrade the Primary

1. Step down the primary to initiate the normal *failover* procedure. Using one of the following:
   - The `rs.stepDown()` helper in the `mongo` shell.
   - The `replSetStepDown` database command.

   During failover, the set cannot accept writes. Typically this takes 10-20 seconds. Plan the upgrade during a predefined maintenance window.

   Note: Stepping down the primary is preferable to directly *shutting down* the primary. Stepping down expedites the failover procedure.

2. Once the primary has stepped down, call the `rs.status()` method from the `mongo` shell until you see that another member has assumed the `PRIMARY` state.
3. Shut down the original primary and upgrade its instance by following the instructions below in *Upgrade a MongoDB Instance* (page 67).

Additional Resources

- MongoDB Major Version Upgrade Consulting Package[6]

2.1.10 Monitor MongoDB With SNMP on Linux

On this page

New in version 2.2.

Enterprise Feature

SNMP is only available in MongoDB Enterprise[7].

[6] https://www.mongodb.com/products/consulting?jmp=docs#major_version_upgrade
[7] http://www.mongodb.com/products/mongodb-enterprise?jmp=docs

Overview

MongoDB Enterprise can provide database metrics via SNMP, in support of centralized data collection and aggregation. This procedure explains the setup and configuration of a `mongod` instance as an SNMP subagent, as well as initializing and testing of SNMP support with MongoDB Enterprise.

See also:

Troubleshoot SNMP (page 73) and *Monitor MongoDB Windows with SNMP* (page 71) for complete instructions on using MongoDB with SNMP on Windows systems.

Considerations

Only `mongod` instances provide SNMP support. `mongos` and the other MongoDB binaries do not support SNMP.

Configuration Files

Changed in version 2.6.

MongoDB Enterprise contains the following configuration files to support SNMP:

- `MONGOD-MIB.txt`:

  The management information base (MIB) file that defines MongoDB's SNMP output.

- `mongod.conf.subagent`:

  The configuration file to run `mongod` as the SNMP subagent. This file sets SNMP run-time configuration options, including the `AgentX` socket to connect to the SNMP master.

- `mongod.conf.master`:

  The configuration file to run `mongod` as the SNMP master. This file sets SNMP run-time configuration options.

Procedure

Step 1: Copy configuration files.

Use the following sequence of commands to move the SNMP configuration files to the SNMP service configuration directory.

First, create the SNMP configuration directory if needed and then, from the installation directory, copy the configuration files to the SNMP service configuration directory:

```
mkdir -p /etc/snmp/
cp MONGOD-MIB.txt /usr/share/snmp/mibs/MONGOD-MIB.txt
cp mongod.conf.subagent /etc/snmp/mongod.conf
```

The configuration filename is tool-dependent. For example, when using `net-snmp` the configuration file is `snmpd.conf`.

By default SNMP uses UNIX domain for communication between the agent (i.e. `snmpd` or the master) and sub-agent (i.e. MongoDB).

Ensure that the `agentXAddress` specified in the SNMP configuration file for MongoDB matches the `agentXAddress` in the SNMP master configuration file.

Step 2: Start MongoDB.

Start `mongod` with the `snmp-subagent` to send data to the SNMP master.

```
mongod --snmp-subagent
```

Step 3: Confirm SNMP data retrieval.

Use `snmpwalk` to collect data from `mongod`:

Connect an SNMP client to verify the ability to collect SNMP data from MongoDB.

Install the net-snmp[8] package to access the `snmpwalk` client. `net-snmp` provides the `snmpwalk` SNMP client.

```
snmpwalk -m /usr/share/snmp/mibs/MONGOD-MIB.txt -v 2c -c mongodb 127.0.0.1:<port> 1.3.6.1.4.1.34
```

`<port>` refers to the port defined by the SNMP master, *not* the primary `port` used by `mongod` for client communication.

Optional: Run MongoDB as SNMP Master

You can run `mongod` with the `snmp-master` option for testing purposes. To do this, use the SNMP master configuration file instead of the subagent configuration file. From the directory containing the unpacked MongoDB installation files:

```
cp mongod.conf.master /etc/snmp/mongod.conf
```

Additionally, start `mongod` with the `snmp-master` option, as in the following:

```
mongod --snmp-master
```

2.1.11 Monitor MongoDB Windows with SNMP

On this page

New in version 2.6.

Enterprise Feature

SNMP is only available in MongoDB Enterprise[9].

[8] http://www.net-snmp.org/
[9] http://www.mongodb.com/products/mongodb-enterprise?jmp=docs

Overview

MongoDB Enterprise can provide database metrics via SNMP, in support of centralized data collection and aggregation. This procedure explains the setup and configuration of a `mongod.exe` instance as an SNMP subagent, as well as initializing and testing of SNMP support with MongoDB Enterprise.

See also:

Monitor MongoDB With SNMP on Linux (page 69) and *Troubleshoot SNMP* (page 73) for more information.

Considerations

Only `mongod.exe` instances provide SNMP support. `mongos.exe` and the other MongoDB binaries do not support SNMP.

Configuration Files

Changed in version 2.6.

MongoDB Enterprise contains the following configuration files to support SNMP:

- `MONGOD-MIB.txt`:

  The management information base (MIB) file that defines MongoDB's SNMP output.

- `mongod.conf.subagent`:

  The configuration file to run `mongod.exe` as the SNMP subagent. This file sets SNMP run-time configuration options, including the `AgentX` socket to connect to the SNMP master.

- `mongod.conf.master`:

  The configuration file to run `mongod.exe` as the SNMP master. This file sets SNMP run-time configuration options.

Procedure

Step 1: Copy configuration files.

Use the following sequence of commands to move the SNMP configuration files to the SNMP service configuration directory.

First, create the SNMP configuration directory if needed and then, from the installation directory, copy the configuration files to the SNMP service configuration directory:

```
md C:\snmp\etc\config
copy MONGOD-MIB.txt C:\snmp\etc\config\MONGOD-MIB.txt
copy mongod.conf.subagent C:\snmp\etc\config\mongod.conf
```

The configuration filename is tool-dependent. For example, when using `net-snmp` the configuration file is `snmpd.conf`.

Edit the configuration file to ensure that the communication between the agent (i.e. `snmpd` or the master) and sub-agent (i.e. MongoDB) uses TCP.

Ensure that the `agentXAddress` specified in the SNMP configuration file for MongoDB matches the `agentXAddress` in the SNMP master configuration file.

Step 2: Start MongoDB.

Start `mongod.exe` with the `snmp-subagent` to send data to the SNMP master.

```
mongod.exe --snmp-subagent
```

Step 3: Confirm SNMP data retrieval.

Use `snmpwalk` to collect data from `mongod.exe`:

Connect an SNMP client to verify the ability to collect SNMP data from MongoDB.

Install the net-snmp[10] package to access the `snmpwalk` client. `net-snmp` provides the `snmpwalk` SNMP client.

```
snmpwalk -m C:\snmp\etc\config\MONGOD-MIB.txt -v 2c -c mongodb 127.0.0.1:<port> 1.3.6.1.4.1.3460
```

`<port>` refers to the port defined by the SNMP master, *not* the primary `port` used by `mongod.exe` for client communication.

Optional: Run MongoDB as SNMP Master

You can run `mongod.exe` with the `snmp-master` option for testing purposes. To do this, use the SNMP master configuration file instead of the subagent configuration file. From the directory containing the unpacked MongoDB installation files:

```
copy mongod.conf.master C:\snmp\etc\config\mongod.conf
```

Additionally, start `mongod.exe` with the `snmp-master` option, as in the following:

```
mongod.exe --snmp-master
```

2.1.12 Troubleshoot SNMP

On this page

New in version 2.6.

Enterprise Feature

SNMP is only available in MongoDB Enterprise.

Overview

MongoDB Enterprise can provide database metrics via SNMP, in support of centralized data collection and aggregation. This document identifies common problems you may encounter when deploying MongoDB Enterprise with SNMP as well as possible solutions for these issues.

[10] http://www.net-snmp.org/

See *Monitor MongoDB With SNMP on Linux* (page 69) and *Monitor MongoDB Windows with SNMP* (page 71) for complete installation instructions.

Issues

Failed to Connect

The following in the `mongod` logfile:

```
Warning: Failed to connect to the agentx master agent
```

AgentX is the SNMP agent extensibility protocol defined in Internet RFC 2741[11]. It explains how to define additional data to monitor over SNMP. When MongoDB fails to connect to the agentx master agent, use the following procedure to ensure that the SNMP subagent can connect properly to the SNMP master.

1. Make sure the master agent is running.
2. Compare the SNMP master's configuration file with the subagent configuration file. Ensure that the agentx socket definition is the same between the two.
3. Check the SNMP configuration files to see if they specify using UNIX Domain Sockets. If so, confirm that the `mongod` has appropriate permissions to open a UNIX domain socket.

Error Parsing Command Line

One of the following errors at the command line:

```
Error parsing command line: unknown option snmp-master
try 'mongod --help' for more information

Error parsing command line: unknown option snmp-subagent
try 'mongod --help' for more information
```

`mongod` binaries that are not part of the Enterprise Edition produce this error. `Install the Enterprise Edition` and attempt to start `mongod` again.

Other MongoDB binaries, including `mongos` will produce this error if you attempt to star them with `snmp-master` or `snmp-subagent`. Only `mongod` supports SNMP.

Error Starting `SNMPAgent`

The following line in the log file indicates that `mongod` cannot read the `mongod.conf` file:

```
[SNMPAgent] warning: error starting SNMPAgent as master err:1
```

If running on Linux, ensure `mongod.conf` exists in the `/etc/snmp` directory, and ensure that the `mongod` UNIX user has permission to read the `mongod.conf` file.

If running on Windows, ensure `mongod.conf` exists in `C:\snmp\etc\config`.

[11] http://www.ietf.org/rfc/rfc2741.txt

2.2 Backup and Recovery

The following tutorials describe backup and restoration for a `mongod` instance:

***Backup and Restore with Filesystem Snapshots* (page 75)** An outline of procedures for creating MongoDB data set backups using system-level file snapshot tool, such as *LVM* or native storage appliance tools.

***Restore a Replica Set from MongoDB Backups* (page 80)** Describes procedure for restoring a replica set from an archived backup such as a `mongodump` or MongoDB Cloud Manager[12] Backup file.

***Back Up and Restore with MongoDB Tools* (page 82)** Describes a procedure for exporting the contents of a database to either a binary dump or a textual exchange format, and for importing these files into a database.

***Backup and Restore Sharded Clusters* (page 88)** Detailed procedures and considerations for backing up sharded clusters and single shards.

***Recover Data after an Unexpected Shutdown* (page 98)** Recover data from MongoDB data files that were not properly closed or have an invalid state.

2.2.1 Backup and Restore with Filesystem Snapshots

On this page

This document describes a procedure for creating backups of MongoDB systems using system-level tools, such as *LVM* or storage appliance, as well as the corresponding restoration strategies.

These filesystem snapshots, or “block-level” backup methods, use system level tools to create copies of the device that holds MongoDB’s data files. These methods complete quickly and work reliably, but require additional system configuration outside of MongoDB.

If the `dbpath` is on a single volume, filesystem snapshots are sufficient for the purpose of volume-level backup of WiredTiger. If you do use journaling, the journal **must** reside on the same volume as the data.

See also:

MongoDB Backup Methods (page 4) and *Back Up and Restore with MongoDB Tools* (page 82).

Snapshots Overview

Snapshots work by creating pointers between the live data and a special snapshot volume. These pointers are theoretically equivalent to “hard links.” As the working data diverges from the snapshot, the snapshot process uses a copy-on-write strategy. As a result the snapshot only stores modified data.

After making the snapshot, you mount the snapshot image on your file system and copy data from the snapshot. The resulting backup contains a full copy of all data.

[12] https://cloud.mongodb.com/?jmp=docs

Considerations

Valid Database at the Time of Snapshot The database must be valid when the snapshot takes place. This means that all writes accepted by the database need to be fully written to disk: either to the *journal* or to data files.

If all writes are not on disk when the backup occurs, the backup will not reflect these changes.

For the `MMAPv1 storage engine`, if writes are *in progress* when the backup occurs, the data files will reflect an inconsistent state. With *journaling* (page 148), all data-file states resulting from in-progress writes are recoverable; without journaling, you must flush all pending writes to disk before running the backup operation and must ensure that no writes occur during the entire backup procedure. If you do use journaling, the journal **must** reside on the same volume as the data.

For the `WiredTiger storage engine`, the data files reflect a consistent state as of the last *checkpoint*, which occurs with every 2 GB of data or every minute.

If the `dbpath` is on a single volume, filesystem snapshots are sufficient for the purpose of volume-level backup of WiredTiger. If you do use journaling, the journal **must** reside on the same volume as the data.

Entire Disk Image Snapshots create an image of an entire disk image. Unless you need to back up your entire system, consider isolating your MongoDB data files, journal (if applicable), and configuration on one logical disk that doesn't contain any other data.

Alternately, store all MongoDB data files on a dedicated device so that you can make backups without duplicating extraneous data.

Site Failure Precaution Ensure that you copy data from snapshots onto other systems. This ensures that data is safe from site failures.

No Incremental Backups This tutorial does not include procedures for incremental backups. Although different snapshots methods provide different capability, the LVM method outlined below does not provide any capacity for capturing incremental backups.

Snapshots With Journaling

If your `mongod` instance has journaling enabled, then you can use any kind of file system or volume/block level snapshot tool to create backups.

If you manage your own infrastructure on a Linux-based system, configure your system with *LVM* to provide your disk packages and provide snapshot capability. You can also use LVM-based setups *within* a cloud/virtualized environment.

Note: Running *LVM* provides additional flexibility and enables the possibility of using snapshots to back up MongoDB.

Snapshots with Amazon EBS in a RAID 10 Configuration

If your deployment depends on Amazon's Elastic Block Storage (EBS) with RAID configured within your instance, it is impossible to get a consistent state across all disks using the platform's snapshot tool. As an alternative, you can do one of the following:

- Flush all writes to disk and create a write lock to ensure consistent state during the backup process.

  If you choose this option see *Create Backups on Instances using MMAPv1 that Do Not Have Journaling Enabled* (page 79).

- Configure *LVM* to run and hold your MongoDB data files on top of the RAID within your system.

  If you choose this option, perform the LVM backup operation described in *Create a Snapshot* (page 77).

Backup and Restore Using LVM on a Linux System

This section provides an overview of a simple backup process using *LVM* on a Linux system. While the tools, commands, and paths may be (slightly) different on your system the following steps provide a high level overview of the backup operation.

Note: Only use the following procedure as a guideline for a backup system and infrastructure. Production backup systems must consider a number of application specific requirements and factors unique to specific environments.

Create a Snapshot

Important: With WiredTiger, the `dbpath`, including the journal files if enabled, must be on a single volume to backup with snapshots.

To create a snapshot with *LVM*, issue a command as root in the following format:

```
lvcreate --size 100M --snapshot --name mdb-snap01 /dev/vg0/mongodb
```

This command creates an *LVM* snapshot (with the `--snapshot` option) named `mdb-snap01` of the `mongodb` volume in the `vg0` volume group.

This example creates a snapshot named `mdb-snap01` located at `/dev/vg0/mdb-snap01`. The location and paths to your systems volume groups and devices may vary slightly depending on your operating system's *LVM* configuration.

The snapshot has a cap of at 100 megabytes, because of the parameter `--size 100M`. This size does not reflect the total amount of the data on the disk, but rather the quantity of differences between the current state of `/dev/vg0/mongodb` and the creation of the snapshot (i.e. `/dev/vg0/mdb-snap01`.)

Warning: Ensure that you create snapshots with enough space to account for data growth, particularly for the period of time that it takes to copy data out of the system or to a temporary image.
If your snapshot runs out of space, the snapshot image becomes unusable. Discard this logical volume and create another.

The snapshot will exist when the command returns. You can restore directly from the snapshot at any time or by creating a new logical volume and restoring from this snapshot to the alternate image.

While snapshots are great for creating high quality backups very quickly, they are not ideal as a format for storing backup data. Snapshots typically depend and reside on the same storage infrastructure as the original disk images. Therefore, it's crucial that you archive these snapshots and store them elsewhere.

Archive a Snapshot

After creating a snapshot, mount the snapshot and copy the data to separate storage. Your system might try to compress the backup images as you move them offline. Alternatively, take a block level copy of the snapshot image, such as with the following procedure:

```
umount /dev/vg0/mdb-snap01
dd if=/dev/vg0/mdb-snap01 | gzip > mdb-snap01.gz
```

The above command sequence does the following:

- Ensures that the `/dev/vg0/mdb-snap01` device is not mounted. Never take a block level copy of a filesystem or filesystem snapshot that is mounted.
- Performs a block level copy of the entire snapshot image using the `dd` command and compresses the result in a gzipped file in the current working directory.

Warning: This command will create a large `gz` file in your current working directory. Make sure that you run this command in a file system that has enough free space.

Restore a Snapshot

To restore a snapshot created with the above method, issue the following sequence of commands:

```
lvcreate --size 1G --name mdb-new vg0
gzip -d -c mdb-snap01.gz | dd of=/dev/vg0/mdb-new
mount /dev/vg0/mdb-new /srv/mongodb
```

The above sequence does the following:

- Creates a new logical volume named `mdb-new`, in the `/dev/vg0` volume group. The path to the new device will be `/dev/vg0/mdb-new`.

  Warning: This volume will have a maximum size of 1 gigabyte. The original file system must have had a total size of 1 gigabyte or smaller, or else the restoration will fail.
  Change `1G` to your desired volume size.

- Uncompresses and unarchives the `mdb-snap01.gz` into the `mdb-new` disk image.
- Mounts the `mdb-new` disk image to the `/srv/mongodb` directory. Modify the mount point to correspond to your MongoDB data file location, or other location as needed.

Note: The restored snapshot will have a stale `mongod.lock` file. If you do not remove this file from the snapshot, and MongoDB may assume that the stale lock file indicates an unclean shutdown. If you're running with `storage.journal.enabled` enabled, and you *do not* use `db.fsyncLock()`, you do not need to remove the `mongod.lock` file. If you use `db.fsyncLock()` you will need to remove the lock.

Restore Directly from a Snapshot

To restore a backup without writing to a compressed `gz` file, use the following sequence of commands:

```
umount /dev/vg0/mdb-snap01
lvcreate --size 1G --name mdb-new vg0
```

```
dd if=/dev/vg0/mdb-snap01 of=/dev/vg0/mdb-new
mount /dev/vg0/mdb-new /srv/mongodb
```

Remote Backup Storage

You can implement off-system backups using the *combined process* (page 78) and SSH.

This sequence is identical to procedures explained above, except that it archives and compresses the backup on a remote system using SSH.

Consider the following procedure:

```
umount /dev/vg0/mdb-snap01
dd if=/dev/vg0/mdb-snap01 | ssh username@example.com gzip > /opt/backup/mdb-snap01.gz
lvcreate --size 1G --name mdb-new vg0
ssh username@example.com gzip -d -c /opt/backup/mdb-snap01.gz | dd of=/dev/vg0/mdb-new
mount /dev/vg0/mdb-new /srv/mongodb
```

Create Backups on Instances using MMAPv1 that Do Not Have Journaling Enabled

If your `mongod` instance is using the `MMAPv1 storage engine` and is either running with journaling disabled or the journal files are on a separate volume, obtaining a functional backup of a consistent state is more complicated. As described in this section, you must flush all writes to disk and lock the database to prevent writes during the backup process. If you have a *replica set* configuration, then for your backup use a *secondary* which is not receiving reads (i.e. *hidden member*).

Important:

- This procedure is only supported with the MMAPv1 storage engine.
- In the following procedure to create backups, you **must** issue the `db.fsyncLock()` and `db.fsyncUnlock()` operations on the same connection. The client that issues `db.fsyncLock()` is solely responsible for issuing a `db.fsyncUnlock()` operation and must be able to handle potential error conditions so that it can perform the `db.fsyncUnlock()` before terminating the connection.

Step 1: Flush writes to disk and lock the database to prevent further writes.

To flush writes to disk and to "lock" the database, issue the `db.fsyncLock()` method in the `mongo` shell:

```
db.fsyncLock();
```

Step 2: Perform the backup operation described in *Create a Snapshot*.

Step 3: After the snapshot completes, unlock the database.

To unlock the database after the snapshot has completed, use the following command in the `mongo` shell:

```
db.fsyncUnlock();
```

Changed in version 2.2: When used in combination with `fsync` or `db.fsyncLock()`, `mongod` will block reads, including those from `mongodump`, when queued write operation waits behind the `fsync` lock. Do not use `mongodump` with `db.fsyncLock()`.

Additional Resources

See also MongoDB Cloud Manager[13] for seamless automation, backup, and monitoring.

2.2.2 Restore a Replica Set from MongoDB Backups

On this page

This procedure outlines the process for taking MongoDB data and restoring that data into a new *replica set*. Use this approach for seeding test deployments from production backups as well as part of disaster recovery.

You *cannot* restore a single data set to three new `mongod` instances and *then* create a replica set. In this situation MongoDB will force the secondaries to perform an initial sync. The procedures in this document describe the correct and efficient ways to deploy a replica set.

Restore Database into a Single Node Replica Set

Step 1: Obtain backup MongoDB Database files.

The backup files may come from a *file system snapshot* (page 75). The MongoDB Cloud Manager[14] produces MongoDB database files for stored snapshots[15] and point in time snapshots[16]. For Ops Manager, an on-premise solution available in MongoDB Enterprise Advanced[17], see also the Ops Manager Backup overview[18].

You can also use `mongorestore` to restore database files using data created with `mongodump`. See *Back Up and Restore with MongoDB Tools* (page 82) for more information.

Step 2: Start a `mongod` using data files from the backup as the data path.

Start a `mongod` instance for a new single-node replica set. Specify the path to the backup data files with `--dbpath` option and the replica set name with the `--replSet` option.

```
mongod --dbpath /data/db --replSet <replName>
```

Step 3: Connect a `mongo` shell to the `mongod` instance.

For example, to connect to a `mongod` running on localhost on the default port of `27017`, simply issue:

```
mongo
```

[13]https://cloud.mongodb.com/?jmp=docs
[14]https://cloud.mongodb.com/?jmp=docs
[15]https://docs.cloud.mongodb.com/tutorial/restore-from-snapshot/
[16]https://docs.cloud.mongodb.com/tutorial/restore-from-point-in-time-snapshot/
[17]https://www.mongodb.com/products/mongodb-enterprise-advanced?jmp=docs
[18]https://docs.opsmanager.mongodb.com/current/core/backup-overview

Step 4: Initiate the new replica set.

Use `rs.initiate()` on the replica set member:

```
rs.initiate()
```

MongoDB initiates a set that consists of the current member and that uses the default replica set configuration.

Add Members to the Replica Set

MongoDB provides two options for restoring secondary members of a replica set:

- Manually copy the database files to each data directory.
- Allow *initial sync* to distribute data automatically.

The following sections outlines both approaches.

Note: If your database is large, initial sync can take a long time to complete. For large databases, it might be preferable to copy the database files onto each host.

Copy Database Files and Restart `mongod` Instance

Use the following sequence of operations to "seed" additional members of the replica set with the restored data by copying MongoDB data files directly.

Step 1: Shut down the `mongod` instance that you restored.

Use `--shutdown` or `db.shutdownServer()` to ensure a clean shut down.

Step 2: Copy the primary's data directory to each secondary.

Copy the *primary's* data directory into the `dbPath` of the other members of the replica set. The `dbPath` is `/data/db` by default.

Step 3: Start the `mongod` instance that you restored.

Step 4: Add the secondaries to the replica set.

In a `mongo` shell connected to the *primary*, add the *secondaries* to the replica set using `rs.add()`. See *Deploy a Replica Set* (page 160) for more information about deploying a replica set.

Update Secondaries using Initial Sync

Use the following sequence of operations to "seed" additional members of the replica set with the restored data using the default *initial sync* operation.

Step 1: Ensure that the data directories on the prospective replica set members are empty.

Step 2: Add each prospective member to the replica set.

When you add a member to the replica set, *Initial Sync* copies the data from the *primary* to the new member.

2.2.3 Back Up and Restore with MongoDB Tools

On this page

This document describes the process for creating backups and restoring data using the utilities provided with MongoDB.

Because all of these tools primarily operate by interacting with a running `mongod` instance, they can impact the performance of your running database.

Not only do they create traffic for a running database instance, they also force the database to read all data through memory. When MongoDB reads infrequently used data, it can supplant more frequently accessed data, causing a deterioration in performance for the database's regular workload.

No matter how you decide to import or export your data, consider the following guidelines:

- Label files so that you can identify the contents of the export or backup as well as the point in time the export/backup reflect.
- Do not create or apply exports if the backup process itself will have an adverse effect on a production system.
- Make sure that the backups reflect a consistent data state. Export or backup processes can impact data integrity (i.e. type fidelity) and consistency if updates continue during the backup process.
- Test backups and exports by restoring and importing to ensure that the backups are useful.

See also:

MongoDB Backup Methods (page 4) or MongoDB Cloud Manager Backup documentation[19] for more information on backing up MongoDB instances. Additionally, consider the following references for the MongoDB import/export tools:

- `mongoimport`
- `mongoexport`
- `mongorestore`
- `mongodump`

Binary BSON Dumps

The `mongorestore` and `mongodump` utilities work with `BSON` data dumps, and are useful for creating backups of small deployments. For resilient and non-disruptive backups, use a file system or block-level disk snapshot function, such as the methods described in the *MongoDB Backup Methods* (page 4) document.

[19]https://docs.cloud.mongodb.com/tutorial/nav/backup-use/

Use these tools for backups if other backup methods, such as the MongoDB Cloud Manager[20] or *file system snapshots* (page 75) are unavailable.

Backup a Database with `mongodump`

`mongodump` does *not* dump the content of the `local` database.

To back up all the databases in a cluster via `mongodump`, you should have the `backup` role. The `backup` role provides the required privileges for backing up all databases. The role confers no additional access, in keeping with the policy of *least privilege*.

To back up a given database, you must have `read` access on the database. Several roles provide this access, including the `backup` role.

To back up the `system.profile` (page 130) collection, which is created when you activate *database profiling* (page 39), you must have **additional** `read` access on this collection. Several roles provide this access, including the `clusterAdmin` and `dbAdmin` roles.

Changed in version 2.6.

To back up users and *user-defined roles* for a given database, you must have access to the `admin` database. MongoDB stores the user data and role definitions for all databases in the `admin` database.

Specifically, to back up a given database's users, you must have the `find` *action* on the `admin` database's `admin.system.users` (page 130) collection. The `backup` and `userAdminAnyDatabase` roles both provide this privilege.

To back up the user-defined roles on a database, you must have the `find` action on the `admin` database's `admin.system.roles` (page 130) collection. Both the `backup` and `userAdminAnyDatabase` roles provide this privilege.

Basic `mongodump` Operations The `mongodump` utility backs up data by connecting to a running `mongod` or `mongos` instance.

The utility can create a backup for an entire server, database or collection, or can use a query to backup just part of a collection.

When you run `mongodump` without any arguments, the command connects to the MongoDB instance on the local system (e.g. `127.0.0.1` or `localhost`) on port `27017` and creates a database backup named `dump/` in the current directory.

To backup data from a `mongod` or `mongos` instance running on the same machine and on the default port of `27017`, use the following command:

```
mongodump
```

The data format used by `mongodump` from version 2.2 or later is *incompatible* with earlier versions of `mongod`. Do not use recent versions of `mongodump` to back up older data stores.

You can also specify the `--host` and `--port` of the MongoDB instance that the `mongodump` should connect to. For example:

```
mongodump --host mongodb.example.net --port 27017
```

`mongodump` will write *BSON* files that hold a copy of data accessible via the `mongod` listening on port `27017` of the `mongodb.example.net` host. See *Create Backups from Non-Local mongod Instances* (page 84) for more information.

[20] https://cloud.mongodb.com/?jmp=docs

To specify a different output directory, you can use the *--out or -o* option:

```
mongodump --out /data/backup/
```

To limit the amount of data included in the database dump, you can specify *--db* and *--collection* as options to `mongodump`. For example:

```
mongodump --collection myCollection --db test
```

This operation creates a dump of the collection named `myCollection` from the database `test` in a `dump/` subdirectory of the current working directory.

`mongodump` overwrites output files if they exist in the backup data folder. Before running the `mongodump` command multiple times, either ensure that you no longer need the files in the output folder (the default is the `dump/` folder) or rename the folders or files.

Point in Time Operation Using Oplogs Use the *--oplog* option with `mongodump` to collect the *oplog* entries to build a point-in-time snapshot of a database within a replica set. With *--oplog*, `mongodump` copies all the data from the source database as well as all of the *oplog* entries from the beginning to the end of the backup procedure. This operation, in conjunction with *mongorestore --oplogReplay*, allows you to restore a backup that reflects the specific moment in time that corresponds to when `mongodump` completed creating the dump file.

Create Backups from Non-Local `mongod` Instances The *--host* and *--port* options for `mongodump` allow you to connect to and backup from a remote host. Consider the following example:

```
mongodump --host mongodb1.example.net --port 3017 --username user --password pass --out /opt/bac
```

On any `mongodump` command you may, as above, specify username and password credentials to specify database authentication.

Restore a Database with `mongorestore`

On systems running with `authorization`, a user must have access that includes the `readWrite` role for each database being restored.

The `readWriteAnyDatabase` role and the `restore` role each provide access to restore any database. If running `mongorestore` with *--oplogReplay*, however, neither role is sufficient. Instead, create a *user-defined role* that has `anyAction` on *resource-anyresource* and grant only to users who must run `mongorestore` with *--oplogReplay*.

Changed in version 2.6.

To restore users and *user-defined roles* on a given database, you must have access to the `admin` database. MongoDB stores the user data and role definitions for all databases in the `admin` database.

Specifically, to restore users to a given database, you must have the `insert` *action* on the `admin` database's `admin.system.users` (page 130) collection. The `restore` role provides this privilege.

To restore user-defined roles to a database, you must have the `insert` action on the `admin` database's `admin.system.roles` (page 130) collection. The `restore` role provides this privilege.

If your database is running with authentication enabled, you must possess the `userAdmin` role on the database you are restoring, or the `userAdminAnyDatabase` role, which allows you to restore user data to any database. The `restore` role also provides the requisite privileges.

Basic `mongorestore` Operations The `mongorestore` utility restores a binary backup created by `mongodump`. By default, `mongorestore` looks for a database backup in the `dump/` directory.

The `mongorestore` utility restores data by connecting to a running `mongod` or `mongos` directly.

`mongorestore` can restore either an entire database backup or a subset of the backup.

To use `mongorestore` to connect to an active `mongod` or `mongos`, use a command with the following prototype form:

```
mongorestore --port <port number> <path to the backup>
```

Consider the following example:

```
mongorestore dump-2013-10-25/
```

Here, `mongorestore` imports the database backup in the `dump-2013-10-25` directory to the `mongod` instance running on the localhost interface.

Restore Point in Time Oplog Backup If you created your database dump using the *`--oplog`* option to ensure a point-in-time snapshot, call `mongorestore` with the *`--oplogReplay`* option, as in the following example:

```
mongorestore --oplogReplay
```

You may also consider using the *`mongorestore --objcheck`* option to check the integrity of objects while inserting them into the database, or you may consider the *`mongorestore --drop`* option to drop each collection from the database before restoring from backups.

Restore Backups to Non-Local `mongod` Instances By default, `mongorestore` connects to a MongoDB instance running on the localhost interface (e.g. `127.0.0.1`) and on the default port (`27017`). If you want to restore to a different host or port, use the *`--host`* and *`--port`* options.

Consider the following example:

```
mongorestore --host mongodb1.example.net --port 3017 --username user --password pass /opt/backup
```

As above, you may specify username and password connections if your `mongod` requires authentication.

Human Intelligible Import/Export Formats

MongoDB's `mongoimport` and `mongoexport` tools allow you to work with your data in a human-readable `Extended JSON` or *CSV* format. This is useful for simple ingestion to or from a third-party system, and when you want to backup or export a small subset of your data. For more complex data migration tasks, you may want to write your own import and export scripts using a client *driver* to interact with the database.

The examples in this section use the MongoDB tools `mongoimport` and `mongoexport`. These tools may also be useful for importing data into a MongoDB database from third party applications.

If you want to simply copy a database or collection from one instance to another, consider using the `copydb`, `clone`, or `cloneCollection` commands, which may be more suited to this task. The `mongo` shell provides the `db.copyDatabase()` method.

Warning:

Warning: Avoid using `mongoimport` and `mongoexport` for full instance production backups. They do not reliably preserve all rich *BSON* data types, because *JSON* can only represent a subset of the types supported by BSON. Use `mongodump` and `mongorestore` as described in *MongoDB Backup Methods* (page 4) for this kind of functionality.

Collection Export with `mongoexport`

Export in CSV Format

Changed in version 3.0.0: `mongoexport` removed the `--csv` option. Use the *--type=csv* option to specify CSV format for the output.

In the following example, `mongoexport` exports data from the collection `contacts` collection in the `users` database in *CSV* format to the file `/opt/backups/contacts.csv`.

The `mongod` instance that `mongoexport` connects to is running on the localhost port number `27017`.

When you export in CSV format, you must specify the fields in the documents to export. The operation specifies the `name` and `address` fields to export.

```
mongoexport --db users --collection contacts --type=csv --fields name,address --out /opt/backups
```

For CSV exports only, you can also specify the fields in a file containing the line-separated list of fields to export. The file must have only one field per line.

For example, you can specify the `name` and `address` fields in a file `fields.txt`:

```
name
address
```

Then, using the *--fieldFile* option, specify the fields to export with the file:

```
mongoexport --db users --collection contacts --type=csv --fieldFile fields.txt --out /opt/backup
```

Changed in version 3.0.0: `mongoexport` removed the `--csv` option and replaced with the *--type* option.

Export in JSON Format

This example creates an export of the `contacts` collection from the MongoDB instance running on the localhost port number `27017`. This writes the export to the `contacts.json` file in *JSON* format.

```
mongoexport --db sales --collection contacts --out contacts.json
```

Export from Remote Host Running with Authentication

The following example exports the `contacts` collection from the `marketing` database, which requires authentication.

This data resides on the MongoDB instance located on the host `mongodb1.example.net` running on port `37017`, which requires the username `user` and the password `pass`.

```
mongoexport --host mongodb1.example.net --port 37017 --username user --password pass --collectio
```

Export Query Results

You can export only the results of a query by supplying a query filter with the *--query* option, and limit the results to a single database using the "*--db*" option.

For instance, this command returns all documents in the sales database's contacts collection that contain a field named field with a value of 1.

```
mongoexport --db sales --collection contacts --query '{"field": 1}'
```

You must enclose the query in single quotes (e.g. ') to ensure that it does not interact with your shell environment.

Collection Import with mongoimport

Simple Usage mongoimport restores a database from a backup taken with mongoexport. Most of the arguments to mongoexport also exist for mongoimport.

In the following example, mongoimport imports the data in the *JSON* data from the contacts.json file into the collection contacts in the users database.

```
mongoimport --db users --collection contacts --file contacts.json
```

Import JSON to Remote Host Running with Authentication In the following example, mongoimport imports data from the file /opt/backups/mdb1-examplenet.json into the contacts collection within the database marketing on a remote MongoDB database with authentication enabled.

mongoimport connects to the mongod instance running on the host mongodb1.example.net over port 37017. It authenticates with the username user and the password pass.

```
mongoimport --host mongodb1.example.net --port 37017 --username user --password pass --collectio
```

CSV Import In the following example, mongoimport imports the *csv* formatted data in the /opt/backups/contacts.csv file into the collection contacts in the users database on the MongoDB instance running on the localhost port numbered 27017.

Specifying *--headerline* instructs mongoimport to determine the name of the fields using the first line in the CSV file.

```
mongoimport --db users --collection contacts --type csv --headerline --file /opt/backups/contact
```

mongoimport uses the input file name, without the extension, as the collection name if -c or --collection is unspecified. The following example is therefore equivalent:

```
mongoimport --db users --type csv --headerline --file /opt/backups/contacts.csv
```

Use the "*--ignoreBlanks*" option to ignore blank fields. For *CSV* and *TSV* imports, this option provides the desired functionality in most cases because it avoids inserting fields with null values into your collection.

Additional Resources

- Backup and its Role in Disaster Recovery White Paper[21]
- Cloud Backup through MongoDB Cloud Manager[22]

[21] https://www.mongodb.com/lp/white-paper/backup-disaster-recovery?jmp=docs
[22] https://cloud.mongodb.com/?jmp=docs

- Blog Post: Backup vs. Replication, Why you Need Both[23]
- Backup Service with Ops Manager, an on-premise solution available in MongoDB Enterprise Advanced[24]

2.2.4 Backup and Restore Sharded Clusters

The following tutorials describe backup and restoration for sharded clusters:

***Backup a Small Sharded Cluster with mongodump* (page 88)** If your *sharded cluster* holds a small data set, you can use `mongodump` to capture the entire backup in a reasonable amount of time.

***Backup a Sharded Cluster with Filesystem Snapshots* (page 90)** Use file system snapshots back up each component in the sharded cluster individually. The procedure involves stopping the cluster balancer. If your system configuration allows file system backups, this might be more efficient than using MongoDB tools.

***Backup a Sharded Cluster with Database Dumps* (page 92)** Create backups using `mongodump` to back up each component in the cluster individually.

***Schedule Backup Window for Sharded Clusters* (page 94)** Limit the operation of the cluster balancer to provide a window for regular backup operations.

***Restore a Single Shard* (page 95)** An outline of the procedure and consideration for restoring a single shard from a backup.

***Restore a Sharded Cluster* (page 96)** An outline of the procedure and consideration for restoring an *entire* sharded cluster from backup.

Backup a Small Sharded Cluster with `mongodump`

On this page

Overview

If your *sharded cluster* holds a small data set, you can connect to a `mongos` using `mongodump`. You can create backups of your MongoDB cluster, if your backup infrastructure can capture the entire backup in a reasonable amount of time and if you have a storage system that can hold the complete MongoDB data set.

See *MongoDB Backup Methods* (page 4) and *Backup and Restore Sharded Clusters* (page 88) for complete information on backups in MongoDB and backups of sharded clusters in particular.

Important: By default `mongodump` issue its queries to the non-primary nodes.

To back up all the databases in a cluster via `mongodump`, you should have the `backup` role. The `backup` role provides the required privileges for backing up all databases. The role confers no additional access, in keeping with the policy of *least privilege*.

[23] http://www.mongodb.com/blog/post/backup-vs-replication-why-do-you-need-both?jmp=docs
[24] https://www.mongodb.com/products/mongodb-enterprise-advanced?jmp=docs

To back up a given database, you must have `read` access on the database. Several roles provide this access, including the `backup` role.

To back up the `system.profile` (page 130) collection, which is created when you activate *database profiling* (page 39), you must have **additional** `read` access on this collection. Several roles provide this access, including the `clusterAdmin` and `dbAdmin` roles.

Changed in version 2.6.

To back up users and *user-defined roles* for a given database, you must have access to the `admin` database. MongoDB stores the user data and role definitions for all databases in the `admin` database.

Specifically, to back up a given database's users, you must have the `find` *action* on the `admin` database's `admin.system.users` (page 130) collection. The `backup` and `userAdminAnyDatabase` roles both provide this privilege.

To back up the user-defined roles on a database, you must have the `find` action on the `admin` database's `admin.system.roles` (page 130) collection. Both the `backup` and `userAdminAnyDatabase` roles provide this privilege.

Considerations

If you use `mongodump` without specifying a database or collection, `mongodump` will capture collection data *and* the cluster meta-data from the *config servers*.

You cannot use the `--oplog` option for `mongodump` when capturing data from `mongos`. As a result, if you need to capture a backup that reflects a single moment in time, you must stop all writes to the cluster for the duration of the backup operation.

Procedure

Capture Data You can perform a backup of a *sharded cluster* by connecting `mongodump` to a `mongos`. Use the following operation at your system's prompt:

```
mongodump --host mongos3.example.net --port 27017
```

`mongodump` will write *BSON* files that hold a copy of data stored in the *sharded cluster* accessible via the `mongos` listening on port `27017` of the `mongos3.example.net` host.

Restore Data Backups created with `mongodump` do not reflect the chunks or the distribution of data in the sharded collection or collections. Like all `mongodump` output, these backups contain separate directories for each database and *BSON* files for each collection in that database.

You can restore `mongodump` output to any MongoDB instance, including a standalone, a *replica set*, or a new *sharded cluster*. When restoring data to sharded cluster, you must deploy and configure sharding before restoring data from the backup. See *Deploy a Sharded Cluster* (page 213) for more information.

Additional Resources

See also MongoDB Cloud Manager[25] for seamless automation, backup, and monitoring.

[25] https://cloud.mongodb.com/?jmp=docs

Backup a Sharded Cluster with Filesystem Snapshots

On this page

Overview

Important: The following procedure applies to the `MMAPv1` storage engine only.

This document describes a procedure for taking a backup of all components of a sharded cluster. This procedure uses file system snapshots to capture a copy of the `mongod` instance. An alternate procedure uses `mongodump` to create binary database dumps when file-system snapshots are not available. See *Backup a Sharded Cluster with Database Dumps* (page 92) for the alternate procedure.

See *MongoDB Backup Methods* (page 4) and *Backup and Restore Sharded Clusters* (page 88) for complete information on backups in MongoDB and backups of sharded clusters in particular.

Important: To capture a point-in-time backup from a sharded cluster you **must** stop *all* writes to the cluster. On a running production system, you can only capture an *approximation* of point-in-time snapshot.

Considerations

Storage Engine The following procedure applies to the `MMAPv1` storage engine only.

Balancing It is *essential* that you stop the balancer before capturing a backup.

If the balancer is active while you capture backups, the backup artifacts may be incomplete and/or have duplicate data, as *chunks* may migrate while recording backups.

Precision In this procedure, you will stop the cluster balancer and take a backup up of the *config database*, and then take backups of each shard in the cluster using a file-system snapshot tool. If you need an exact moment-in-time snapshot of the system, you will need to stop all application writes before taking the filesystem snapshots; otherwise the snapshot will only approximate a moment in time.

For approximate point-in-time snapshots, you can improve the quality of the backup while minimizing impact on the cluster by taking the backup from a secondary member of the replica set that provides each shard.

Consistency If the journal and data files are on the same logical volume, you can use a single point-in-time snapshot to capture a valid copy of the data.

If the journal and data files are on different file systems, you must use `db.fsyncLock()` and `db.fsyncUnLock()` to capture a valid copy of your data.

Procedure

Important: The following procedure, which includes `db.fsyncLock()` and `db.fsyncUnlock()` operations, applies only to MongoDB instances using MMAPv1 storage engine.

Step 1: Disable the balancer. Disable the *balancer* process that equalizes the distribution of data among the *shards*. To disable the balancer, use the `sh.stopBalancer()` method in the `mongo` shell.

Consider the following example:

```
use config
sh.stopBalancer()
```

For more information, see the *Disable the Balancer* (page 240) procedure.

Step 2: If necessary, lock one secondary member of each replica set in each shard. If your `mongod` does not have journaling enabled *or* your journal and data files are on different volumes, you **must** lock your `mongod` before capturing a back up.

If your `mongod` has journaling enabled and your journal and data files are on the same volume, you may skip this step.

If you need to lock the `mongod`, attempt to lock one secondary member of each replica set in each shard so that your backups reflect the state of your database at the nearest possible approximation of a single moment in time.

To lock a secondary, connect through the `mongo` shell to the secondary member's `mongod` instance and issue the `db.fsyncLock()` method.

Step 3: Back up one of the config servers. Backing up a *config server* backs up the sharded cluster's metadata. You need back up only one config server, as they all hold the same data. Do one of the following to back up one of the config servers:

Create a file-system snapshot of the config server. Do this **only if** the config server has *journaling* enabled. Use the procedure in *Backup and Restore with Filesystem Snapshots* (page 75). **Never** use `db.fsyncLock()` on config databases.

Create a database dump to backup the config server. Issue `mongodump` against one of the config `mongod` instances. If you are running MongoDB 2.4 or later with the *`--configsvr`* option, then include the *`--oplog`* option to ensure that the dump includes a partial oplog containing operations from the duration of the mongodump operation. For example:

```
mongodump --oplog
```

Step 4: Back up the replica set members of the shards that you locked. You may back up the shards in parallel. For each shard, create a snapshot. Use the procedure in *Backup and Restore with Filesystem Snapshots* (page 75).

Step 5: Unlock locked replica set members. If you locked any `mongod` instances to capture the backup, unlock them now.

Unlock all locked replica set members of each shard using the `db.fsyncUnlock()` method in the `mongo` shell.

Step 6: Enable the balancer. Re-enable the balancer with the `sh.setBalancerState()` method. Use the following command sequence when connected to the `mongos` with the `mongo` shell:

```
use config
sh.setBalancerState(true)
```

Additional Resources

See also MongoDB Cloud Manager[26] for seamless automation, backup, and monitoring.

Backup a Sharded Cluster with Database Dumps

On this page

Overview

Important: The following procedure applies to MongoDB instances using the MMAPv1 storage engine.

This document describes a procedure for taking a backup of all components of a sharded cluster. This procedure uses `mongodump` to create dumps of the `mongod` instance. An alternate procedure uses file system snapshots to capture the backup data, and may be more efficient in some situations if your system configuration allows file system backups. See *Backup and Restore Sharded Clusters* (page 88) for more information.

See *MongoDB Backup Methods* (page 4) and *Backup and Restore Sharded Clusters* (page 88) for complete information on backups in MongoDB and backups of sharded clusters in particular.

Prerequisites

Important: To capture a point-in-time backup from a sharded cluster you **must** stop *all* writes to the cluster. On a running production system, you can only capture an *approximation* of point-in-time snapshot.

To back up all the databases in a cluster via `mongodump`, you should have the `backup` role. The `backup` role provides the required privileges for backing up all databases. The role confers no additional access, in keeping with the policy of *least privilege*.

To back up a given database, you must have `read` access on the database. Several roles provide this access, including the `backup` role.

To back up the `system.profile` (page 130) collection, which is created when you activate *database profiling* (page 39), you must have **additional** `read` access on this collection. Several roles provide this access, including the `clusterAdmin` and `dbAdmin` roles.

[26] https://cloud.mongodb.com/?jmp=docs

Changed in version 2.6.

To back up users and *user-defined roles* for a given database, you must have access to the `admin` database. MongoDB stores the user data and role definitions for all databases in the `admin` database.

Specifically, to back up a given database's users, you must have the `find` *action* on the `admin` database's `admin.system.users` (page 130) collection. The `backup` and `userAdminAnyDatabase` roles both provide this privilege.

To back up the user-defined roles on a database, you must have the `find` action on the `admin` database's `admin.system.roles` (page 130) collection. Both the `backup` and `userAdminAnyDatabase` roles provide this privilege.

Consideration

To create these backups of a sharded cluster, you will stop the cluster balancer and take a backup of the *config database*, and then take backups of each shard in the cluster using `mongodump` to capture the backup data. To capture a more exact moment-in-time snapshot of the system, you will need to stop all application writes before taking the filesystem snapshots; otherwise the snapshot will only approximate a moment in time.

For approximate point-in-time snapshots, taking the backup from a single offline secondary member of the replica set that provides each shard can improve the quality of the backup while minimizing impact on the cluster.

Procedure

Important: The following procedure, which includes `db.fsyncLock()` and `db.fsyncUnlock()` operations, applies only to MongoDB instances using MMAPv1 storage engine.

Step 1: Disable the balancer process. Disable the *balancer* process that equalizes the distribution of data among the *shards*. To disable the balancer, use the `sh.stopBalancer()` method in the `mongo` shell. For example:

```
use config
sh.setBalancerState(false)
```

For more information, see the *Disable the Balancer* (page 240) procedure.

Warning: If you do not stop the balancer, the backup could have duplicate data or omit data as *chunks* migrate while recording backups.

Step 2: Lock replica set members. Lock one member of each replica set in each shard so that your backups reflect the state of your database at the nearest possible approximation of a single moment in time. Lock these `mongod` instances in as short of an interval as possible.

To lock or freeze a sharded cluster, issue `db.fsyncLock()` on a member of each replica set in the cluster. Ensure that the *oplog* has sufficient capacity to allow these secondaries to catch up to the state of the primaries after finishing the backup procedure. See *replica-set-oplog-sizing* for more information.

Step 3: Backup one config server. Run `mongodump` against a config server `mongod` instance to back up the cluster's metadata. The config server `mongod` instance must be version 2.4 or later and must run with the *--configsvr* option. You only need to back up one config server.

Use `mongodump` with the *--oplog* option to backup one of the *config servers*.

```
mongodump --oplog
```

Step 4: Backup replica set members. Back up the "frozen" replica set members of the shards using `mongodump` and specifying the *--oplog* option. You may back up the shards in parallel. Consider the following invocation:

```
mongodump --oplog --out /data/backup/
```

You must run `mongodump` on the same system where the `mongod` ran. `mongodump` writes the output of this dump as well as the `oplog.bson` file to the `/data/backup/` directory.

Step 5: Unlock replica set members. Use `db.fsyncUnlock()` to unlock the locked replica set members of each shard. Allow these members to catch up with the state of the primary.

Step 6: Re-enable the balancer process. Re-enable the balancer with the `sh.setBalancerState()` method.

Use the following command sequence when connected to the `mongos` with the `mongo` shell:

```
use config
sh.setBalancerState(true)
```

Additional Resources

See also MongoDB Cloud Manager[27] for seamless automation, backup, and monitoring.

Schedule Backup Window for Sharded Clusters

On this page

Overview

In a *sharded cluster*, the balancer process is responsible for distributing sharded data around the cluster, so that each *shard* has roughly the same amount of data.

However, when creating backups from a sharded cluster it is important that you disable the balancer while taking backups to ensure that no chunk migrations affect the content of the backup captured by the backup procedure. Using the procedure outlined in the section *Disable the Balancer* (page 240) you can manually stop the balancer process temporarily. As an alternative you can use this procedure to define a balancing window so that the balancer is always disabled during your automated backup operation.

[27] https://cloud.mongodb.com/?jmp=docs

Procedure

If you have an automated backup schedule, you can disable all balancing operations for a period of time. For instance, consider the following command:

```
use config
db.settings.update( { _id : "balancer" }, { $set : { activeWindow : { start : "6:00", stop : "23
```

This operation configures the balancer to run between 6:00am and 11:00pm, server time. Schedule your backup operation to run *and complete* outside of this time. Ensure that the backup can complete outside the window when the balancer is running *and* that the balancer can effectively balance the collection among the shards in the window allotted to each.

Restore a Single Shard

On this page

Overview

Restoring a single shard from backup with other unaffected shards requires a number of special considerations and practices. This document outlines the additional tasks you must perform when restoring a single shard.

Consider the following resources on backups in general as well as backup and restoration of sharded clusters specifically:

- *Backup and Restore Sharded Clusters* (page 88)
- *Restore a Sharded Cluster* (page 96)
- *MongoDB Backup Methods* (page 4)

Procedure

Always restore *sharded clusters* as a whole. When you restore a single shard, keep in mind that the *balancer* process might have moved *chunks* to or from this shard since the last backup. If that's the case, you must manually move those chunks, as described in this procedure.

Step 1: Restore the shard as you would any other `mongod` instance. See *MongoDB Backup Methods* (page 4) for overviews of these procedures.

Step 2: Manage the chunks. For all chunks that migrate away from this shard, you do not need to do anything at this time. You do not need to delete these documents from the shard because the chunks are automatically filtered out from queries by `mongos`. You can remove these documents from the shard, if you like, at your leisure.

For chunks that migrate to this shard after the most recent backup, you must manually recover the chunks using backups of other shards, or some other source. To determine what chunks have moved, view the `changelog` collection in the *config-database*.

Restore a Sharded Cluster

On this page

Overview

You can restore a sharded cluster either from *snapshots* (page 75) or from *BSON database dumps* (page 92) created by the mongodump tool. This document describes procedures to

- *Restore a Sharded Cluster with Filesystem Snapshots* (page 96)
- *Restore a Sharded Cluster with Database Dumps* (page 97)

Procedures

Restore a Sharded Cluster with Filesystem Snapshots The following procedure outlines the steps to restore a sharded cluster from filesystem snapshots. For information on using filesystem snapshots to backup sharded clusters, see *Backup a Sharded Cluster with Filesystem Snapshots* (page 90).

Step 1: Shut down the entire cluster. Stop all mongos and mongod processes, including all shards *and* all config servers. To stop all members, connect to **each** member and issue following operations:

```
use admin
db.shutdownServer()
```

For version 2.4 or earlier, use `db.shutdownServer({force:true})`.

Step 2: Restore the data files. On each server, extract the data files to the location where the mongod instance will access them and restore the following:

- Data files for each server in each shard.

  Because each production *shard* is a replica set, for each shard, restore all the members of the replica set. See *Restore a Replica Set from MongoDB Backups* (page 80).

- Data files for each config server.

See also:

Restore a Snapshot (page 78)

Step 3: Restart the config servers. Restart each *config server* mongod instance by issuing a command similar to the following for each, using values appropriate to your configuration:

```
mongod --configsvr --dbpath /data/configdb --port 27019
```

Step 4: Start one `mongos` instance. Start **one** mongos instance. For the `--configdb`, specify the hostnames (and port numbers) of the config servers started in the step Restart the config servers. (page **??**)

Step 5: If shard hostnames have changed, update the config database. If shard hostnames **have changed**, update the `shards` collection in the *config-database* to reflect the new hostnames.

Step 6: Restart all the shard `mongod` instances.

Step 7: Restart the other `mongos` instances. Restart the remaining two `mongos` instances. For the `--configdb`, specify the hostnames (and port numbers) of the config servers started in the step Restart the config servers. (page **??**)

Step 8: Verify that the cluster is operational. Connect to a `mongos` instance from a `mongo` shell and use the `db.printShardingStatus()` method to ensure that the cluster is operational, as follows:

```
db.printShardingStatus()
show collections
```

Restore a Sharded Cluster with Database Dumps Changed in version 3.0: `mongorestore` requires a running MongoDB instances. Earlier versions of `mongorestore` did not require a running MongoDB instances and instead used the `--dbpath` option. For instructions specific to your version of `mongorestore`, refer to the appropriate version of the manual.

The following procedure outlines the steps to restore a sharded cluster from the BSON database dumps created by `mongodump`. For information on using `mongodump` to backup sharded clusters, see *Backup a Sharded Cluster with Database Dumps* (page 92)

The procedure deploys a new sharded cluster and restores data from database dumps.

Step 1: Deploy a new replica set for each shard. For each shard, deploy a new replica set:

1.Start a new `mongod` for each member of the replica set. Include any other configuration as appropriate.

2.Connect a `mongo` to *one* of the `mongod` instances. In the `mongo` shell:

   (a)Run `rs.initiate()`.

   (b)Use `rs.add()` to add the other members of the replica set.

For detailed instructions on deploying a replica set, see *Deploy a Replica Set* (page 160).

Step 2: Deploy three new config servers. Start three `mongod` instances for the config servers (i.e. `mongod --configsvr`). Include any other configuration as appropriate.

For detailed instructions on setting up the config servers, see *Start the Config Server Database Instances* (page 213).

Step 3: Start the `mongos` instances. Start the `mongos` instances, specifying the new config servers with `--configdb`. Include any other configuration as appropriate.

For detailed instructions on starting the `mongos` instances for a sharded cluster, see *Start the mongos Instances* (page 214).

Step 4: Add shards to the cluster. Connect a `mongo` shell to a `mongos` instance. Use `sh.addShard()` to add each replica sets as a shard.

For detailed instructions in adding shards to the cluster, see *Add Shards to the Cluster* (page 214).

Step 5: Shut down the mongos instances. Once the new sharded cluster is up, shut down all `mongos` instances.

Step 6: Restore the shard data. For each shard, use `mongorestore` to restore the data dump to the primary's data directory. Include the `--drop` option to drop the collections before restoring and, because the *backup procedure* (page 92) included the *`--oplog`* option, include the *`--oplogReplay`* option for `mongorestore`.

For example, on the primary for ShardA, run the `mongorestore`. Specify any other configuration as appropriate.

```
mongorestore --drop --oplogReplay /data/dump/shardA
```

After you have finished restoring all the shards, shut down all shard instances.

Step 7: Restore the config server data. For each config server, use `mongorestore` to restore the data dump to each config server's data directory. Include the `--drop` option to drop the collections before restoring and, because the *backup procedure* (page 92) included the *`--oplog`* option, include the *`--oplogReplay`* option for `mongorestore`.

```
mongorestore --drop --oplogReplay /data/dump/configData
```

Step 8: Start one mongos instance. Start **one** `mongos` instance. For the `--configdb`, specify the hostnames (and port numbers) of the config servers started in the step Deploy three new config servers. (page **??**)

Step 9: If shard hostnames have changed, update the config database. If shard hostnames **have changed**, update the `shards` collection in the *config-database* to reflect the new hostnames.

Step 10: Restart all the shard mongod instances.

Step 11: Restart the other mongos instances. Restart the remaining two `mongos` instances. For the `--configdb`, specify the hostnames (and port numbers) of the config servers started in the step Deploy three new config servers. (page **??**)

Step 12: Verify that the cluster is operational. Connect to a `mongos` instance from a `mongo` shell and use the `db.printShardingStatus()` method to ensure that the cluster is operational, as follows:

```
db.printShardingStatus()
show collections
```

See also:

MongoDB Backup Methods (page 4), *Backup and Restore Sharded Clusters* (page 88)

2.2.5 Recover Data after an Unexpected Shutdown

On this page

If MongoDB does not shutdown cleanly, the on-disk representation of the data files will likely reflect an inconsistent state which could lead to data corruption. [28]

To prevent data inconsistency and corruption, always shut down the database cleanly and use the *durability journaling*. MongoDB writes data to the journal, by default, every 100 milliseconds, such that MongoDB can always recover to a consistent state even in the case of an unclean shutdown due to power loss or other system failure.

If you are *not* running as part of a *replica set* **and** do *not* have journaling enabled, use the following procedure to recover data that may be in an inconsistent state. If you are running as part of a replica set, you should *always* restore from a backup or restart the `mongod` instance with an empty `dbPath` and allow MongoDB to perform an initial sync to restore the data.

To ensure a clean shut down, use one of the following methods:

- `db.shutdownServer()` from the `mongo` shell,
- Your system's *init script*,
- "Control-C" when running `mongod` in interactive mode,
- `kill $(pidof mongod)`; or `kill -2 $(pidof mongod)`,
- On Linux, the *`mongod --shutdown`* option.

See also:

The `https://docs.mongodb.org/manual/administration` documents, including *Replica Set Syncing*, and the documentation on the *`--repair`* `repairPath` and `storage.journal.enabled` settings.

Process

Indications

When you are aware of a `mongod` instance running without journaling that stops unexpectedly **and** you're not running with replication, you should always run the repair operation before starting MongoDB again. If you're using replication, then restore from a backup and allow replication to perform an initial *sync* to restore data.

If the `mongod.lock` file in the data directory specified by `dbPath`, `/data/db` by default, is *not* a zero-byte file, then `mongod` will refuse to start, and you will find a message that contains the following line in your MongoDB log our output:

```
Unclean shutdown detected.
```

This indicates that you need to run `mongod` with the *`--repair`* option. If you run repair when the `mongodb.lock` file exists in your `dbPath`, or the optional *`--repairpath`*, you will see a message that contains the following line:

[28] You can also use the `db.collection.validate()` method to test the integrity of a single collection. However, this process is time consuming, and without journaling you can safely assume that the data is in an invalid state and you should either run the repair operation or resync from an intact member of the replica set.

```
old lock file: /data/db/mongod.lock. probably means unclean shutdown
```

If you see this message, as a last resort you may remove the lockfile **and** run the repair operation before starting the database normally, as in the following procedure:

Overview

Warning: Recovering a member of a replica set.
Do not use this procedure to recover a member of a *replica set*. Instead you should either restore from a *backup* (page 4) or perform an initial sync using data from an intact member of the set, as described in *Resync a Member of a Replica Set* (page 193).

There are two processes to repair data files that result from an unexpected shutdown:

- Use the `--repair` option in conjunction with the `--repairpath` option. `mongod` will read the existing data files, and write the existing data to new data files.

  You do not need to remove the `mongod.lock` file before using this procedure.

- Use the `--repair` option. `mongod` will read the existing data files, write the existing data to new files and replace the existing, possibly corrupt, files with new files.

  You must remove the `mongod.lock` file before using this procedure.

Note: `--repair` functionality is also available in the shell with the `db.repairDatabase()` helper for the `repairDatabase` command.

Procedures

Important: Always Run `mongod` as the same user to avoid changing the permissions of the MongoDB data files.

Repair Data Files and Preserve Original Files

To repair your data files using the `--repairpath` option to preserve the original data files unmodified.

Step 1: Start `mongod` using the option to replace the original files with the repaired files. Start the `mongod` instance using the `--repair` option **and** the `--repairpath` option. Issue a command similar to the following:

```
mongod --dbpath /data/db --repair --repairpath /data/db0
```

When this completes, the new repaired data files will be in the `/data/db0` directory.

Step 2: Start `mongod` with the new data directory. Start `mongod` using the following invocation to point the `dbPath` at `/data/db0`:

```
mongod --dbpath /data/db0
```

Once you confirm that the data files are operational you may delete or archive the old data files in the `/data/db` directory. You may also wish to move the repaired files to the old database location or update the `dbPath` to indicate the new location.

Repair Data Files without Preserving Original Files

To repair your data files without preserving the original files, do not use the *`--repairpath`* option, as in the following procedure:

> **Warning:** After you remove the `mongod.lock` file you *must* run the *`--repair`* process before using your database.

Step 1: Remove the stale lock file. For example:

```
rm /data/db/mongod.lock
```

Replace `/data/db` with your `dbPath` where your MongoDB instance's data files reside.

Step 2: Start `mongod` using the option to replace the original files with the repaired files. Start the `mongod` instance using the *`--repair`* option, which replaces the original data files with the repaired data files. Issue a command similar to the following:

```
mongod --dbpath /data/db --repair
```

When this completes, the repaired data files will replace the original data files in the `/data/db` directory.

Step 3: Start `mongod` as usual. Start `mongod` using the following invocation to point the `dbPath` at `/data/db`:

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

`mongod.lock`

In normal operation, you should **never** remove the `mongod.lock` file and start `mongod`. Instead consider the one of the above methods to recover the database and remove the lock files. In dire situations you can remove the lockfile, and start the database using the possibly corrupt files, and attempt to recover data from the database; however, it's impossible to predict the state of the database in these situations.

If you are not running with journaling, and your database shuts down unexpectedly for *any* reason, you should always proceed *as if* your database is in an inconsistent and likely corrupt state. If at all possible restore from *backup* (page 4) or, if running as a *replica set*, restore by performing an initial sync using data from an intact member of the set, as described in *Resync a Member of a Replica Set* (page 193).

2.3 MongoDB Scripting

The `mongo` shell is an interactive JavaScript shell for MongoDB, and is part of all MongoDB distributions[29]. This section provides an introduction to the shell, and outlines key functions, operations, and use of the `mongo`

[29] http://www.mongodb.org/downloads

shell. Also consider `https://docs.mongodb.org/manual/faq/mongo` and the `shell method` and other relevant `reference material`.

Note: Most examples in the `MongoDB Manual` use the `mongo` shell; however, many `drivers` provide similar interfaces to MongoDB.

***Server-side JavaScript* (page 102)** Details MongoDB's support for executing JavaScript code for server-side operations.

***Data Types in the mongo Shell* (page 104)** Describes the super-set of JSON available for use in the `mongo` shell.

***Write Scripts for the mongo Shell* (page 106)** An introduction to the `mongo` shell for writing scripts to manipulate data and administer MongoDB.

***Getting Started with the mongo Shell* (page 109)** Introduces the use and operation of the MongoDB shell.

***Access the mongo Shell Help Information* (page 113)** Describes the available methods for accessing online help for the operation of the `mongo` interactive shell.

***mongo Shell Quick Reference* (page 115)** A high level reference to the use and operation of the `mongo` shell.

2.3.1 Server-side JavaScript

On this page

Overview

MongoDB provides the following commands, methods, and operator that perform server-side execution of JavaScript code:

- `mapReduce` and the corresponding `mongo` shell method `db.collection.mapReduce()`. `mapReduce` operations *map*, or associate, values to keys, and for keys with multiple values, *reduce* the values for each key to a single object. For more information, see `https://docs.mongodb.org/manual/core/map-reduce`.
- `$where` operator that evaluates a JavaScript expression or a function in order to query for documents.

You can also specify a JavaScript file to the `mongo` shell to run on the server. For more information, see *Running .js files via a mongo shell Instance on the Server* (page 103)

JavaScript in MongoDB

Although these methods use JavaScript, most interactions with MongoDB do not use JavaScript but use an `idiomatic driver` in the language of the interacting application.

You can also disable server-side execution of JavaScript. For details, see *Disable Server-Side Execution of JavaScript* (page 103).

Running `.js` files via a `mongo` shell Instance on the Server

You can specify a JavaScript (`.js`) file to a `mongo` shell instance to execute the file on the server. This is a good technique for performing batch administrative work. When you run `mongo` shell on the server, connecting via the localhost interface, the connection is fast with low latency.

The *command helpers* (page 116) provided in the `mongo` shell are not available in JavaScript files because they are not valid JavaScript. The following table maps the most common `mongo` shell helpers to their JavaScript equivalents.

Shell Helpers	JavaScript Equivalents
`show dbs`, `show databases`	`db.adminCommand('listDatabases')`
`use <db>`	`db = db.getSiblingDB('<db>')`
`show collections`	`db.getCollectionNames()`
`show users`	`db.getUsers()`
`show roles`	`db.getRoles({showBuiltinRoles: true})`
`show log <logname>`	`db.adminCommand({ 'getLog' : '<logname>' })`
`show logs`	`db.adminCommand({ 'getLog' : '*' })`

`it`	`cursor = db.collection.find()`
	`if ( cursor.hasNext() ){`
	`   cursor.next();`
	`}`

Concurrency

Changed in version 2.4.

The V8 JavaScript engine, which became the default in 2.4, allows multiple JavaScript operations to execute at the same time. Prior to 2.4, MongoDB operations that required the JavaScript interpreter had to acquire a lock, and a single `mongod` could only run a single JavaScript operation at a time.

Refer to the individual method or operator documentation for any concurrency information. See also the *concurrency table*.

Disable Server-Side Execution of JavaScript

You can disable all server-side execution of JavaScript, by passing the `--noscripting` option on the command line or setting `security.javascriptEnabled` in a configuration file.

See also:

Store a JavaScript Function on the Server (page 65)

2.3.2 Data Types in the `mongo` Shell

On this page

MongoDB *BSON* provides support for additional data types than *JSON*. `Drivers` provide native support for these data types in host languages and the `mongo` shell also provides several helper classes to support the use of these data types in the `mongo` JavaScript shell. See the `Extended JSON` reference for additional information.

Types

Date

The `mongo` shell provides various methods to return the date, either as a string or as a `Date` object:

- `Date()` method which returns the current date as a string.
- `new Date()` constructor which returns a `Date` object using the `ISODate()` wrapper.
- `ISODate()` constructor which returns a `Date` object using the `ISODate()` wrapper.

Internally, *document-bson-type-date* objects are stored as a 64 bit integer representing the number of milliseconds since the Unix epoch (Jan 1, 1970), which results in a representable date range of about 290 millions years into the past and future.

Return Date as a String To return the date as a string, use the `Date()` method, as in the following example:

```
var myDateString = Date();
```

To print the value of the variable, type the variable name in the shell, as in the following:

```
myDateString
```

The result is the value of `myDateString`:

```
Wed Dec 19 2012 01:03:25 GMT-0500 (EST)
```

To verify the type, use the `typeof` operator, as in the following:

```
typeof myDateString
```

The operation returns `string`.

Return `Date` The `mongo` shell wraps objects of `Date` type with the `ISODate` helper; however, the objects remain of type `Date`.

The following example uses both the `new Date()` constructor and the `ISODate()` constructor to return `Date` objects.

```
var myDate = new Date();
var myDateInitUsingISODateWrapper = ISODate();
```

You can use the `new` operator with the `ISODate()` constructor as well.

To print the value of the variable, type the variable name in the shell, as in the following:

```
myDate
```

The result is the `Date` value of `myDate` wrapped in the `ISODate()` helper:

```
ISODate("2012-12-19T06:01:17.171Z")
```

To verify the type, use the `instanceof` operator, as in the following:

```
myDate instanceof Date
myDateInitUsingISODateWrapper instanceof Date
```

The operation returns `true` for both.

ObjectId

The `mongo` shell provides the `ObjectId()` wrapper class around the *ObjectId* data type. To generate a new ObjectId, use the following operation in the `mongo` shell:

```
new ObjectId
```

See

`https://docs.mongodb.org/manual/reference/object-id` for full documentation of ObjectIds in MongoDB.

NumberLong

By default, the `mongo` shell treats all numbers as floating-point values. The `mongo` shell provides the `NumberLong()` wrapper to handle 64-bit integers.

The `NumberLong()` wrapper accepts the long as a string:

```
NumberLong("2090845886852")
```

The following examples use the `NumberLong()` wrapper to write to the collection:

```
db.collection.insert( { _id: 10, calc: NumberLong("2090845886852") } )
db.collection.update( { _id: 10 },
                      { $set:  { calc: NumberLong("2555555000000") } } )
db.collection.update( { _id: 10 },
                      { $inc: { calc: NumberLong(5) } } )
```

Retrieve the document to verify:

```
db.collection.findOne( { _id: 10 } )
```

In the returned document, the `calc` field contains a `NumberLong` object:

```
{ "_id" : 10, "calc" : NumberLong("2555555000005") }
```

If you use the `$inc` to increment the value of a field that contains a `NumberLong` object by a **float**, the data type changes to a floating point value, as in the following example:

1. Use `$inc` to increment the `calc` field by `5`, which the `mongo` shell treats as a float:

   ```
   db.collection.update( { _id: 10 },
                         { $inc: { calc: 5 } } )
   ```

2. Retrieve the updated document:

   ```
   db.collection.findOne( { _id: 10 } )
   ```

   In the updated document, the `calc` field contains a floating point value:

   ```
   { "_id" : 10, "calc" : 2555555000010 }
   ```

NumberInt

By default, the `mongo` shell treats all numbers as floating-point values. The `mongo` shell provides the `NumberInt()` constructor to explicitly specify 32-bit integers.

Check Types in the `mongo` Shell

To determine the type of fields, the `mongo` shell provides the `instanceof` and `typeof` operators.

`instanceof`

`instanceof` returns a boolean to test if a value is an instance of some type.

For example, the following operation tests whether the `_id` field is an instance of type `ObjectId`:

```
mydoc._id instanceof ObjectId
```

The operation returns `true`.

`typeof`

`typeof` returns the type of a field.

For example, the following operation returns the type of the `_id` field:

```
typeof mydoc._id
```

In this case `typeof` will return the more generic `object` type rather than `ObjectId` type.

2.3.3 Write Scripts for the `mongo` Shell

On this page

You can write scripts for the mongo shell in JavaScript that manipulate data in MongoDB or perform administrative operation. For more information about the mongo shell see *MongoDB Scripting* (page 101), and see the *Running .js files via a mongo shell Instance on the Server* (page 103) section for more information about using these mongo script.

This tutorial provides an introduction to writing JavaScript that uses the mongo shell to access MongoDB.

Opening New Connections

From the mongo shell or from a JavaScript file, you can instantiate database connections using the Mongo() constructor:

```
new Mongo()
new Mongo(<host>)
new Mongo(<host:port>)
```

Consider the following example that instantiates a new connection to the MongoDB instance running on localhost on the default port and sets the global db variable to myDatabase using the getDB() method:

```
conn = new Mongo();
db = conn.getDB("myDatabase");
```

If connecting to a MongoDB instance that has enforces access control, you can use the db.auth() method to authenticate.

Additionally, you can use the connect() method to connect to the MongoDB instance. The following example connects to the MongoDB instance that is running on localhost with the non-default port 27020 and set the global db variable:

```
db = connect("localhost:27020/myDatabase");
```

See also:

https://docs.mongodb.org/manual/reference/method/

Differences Between Interactive and Scripted mongo

When writing scripts for the mongo shell, consider the following:

- To set the db global variable, use the getDB() method or the connect() method. You can assign the database reference to a variable other than db.
- Write operations in the mongo shell use a write concern of *{ w: 1 }* by default. If performing bulk operations, use the Bulk() methods. See *write-methods-incompatibility* for more information.

  Changed in version 2.6: Before MongoDB 2.6, call db.getLastError() explicitly to wait for the result of write operations.
- You **cannot** use any shell helper (e.g. use <dbname>, show dbs, etc.) inside the JavaScript file because they are not valid JavaScript.

  The following table maps the most common mongo shell helpers to their JavaScript equivalents.

Shell Helpers	JavaScript Equivalents
`show dbs`, `show databases`	`db.adminCommand('listDatabases')`
`use <db>`	`db = db.getSiblingDB('<db>')`
`show collections`	`db.getCollectionNames()`
`show users`	`db.getUsers()`
`show roles`	`db.getRoles({showBuiltinRoles: true})`
`show log <logname>`	`db.adminCommand({ 'getLog' : '<logname>' })`
`show logs`	`db.adminCommand({ 'getLog' : '*' })`

`it`	`cursor = db.collection.find()`
	`if ( cursor.hasNext() ){`
	`   cursor.next();`
	`}`

- In interactive mode, `mongo` prints the results of operations including the content of all cursors. In scripts, either use the JavaScript `print()` function or the `mongo` specific `printjson()` function which returns formatted JSON.

Example

To print all items in a result cursor in `mongo` shell scripts, use the following idiom:

```
cursor = db.collection.find();
while ( cursor.hasNext() ) {
   printjson( cursor.next() );
}
```

Scripting

From the system prompt, use `mongo` to evaluate JavaScript.

`--eval` option

Use the `--eval` option to `mongo` to pass the shell a JavaScript fragment, as in the following:

```
mongo test --eval "printjson(db.getCollectionNames())"
```

This returns the output of `db.getCollectionNames()` using the `mongo` shell connected to the `mongod` or `mongos` instance running on port `27017` on the `localhost` interface.

Execute a JavaScript file

You can specify a `.js` file to the `mongo` shell, and `mongo` will execute the JavaScript directly. Consider the following example:

```
mongo localhost:27017/test myjsfile.js
```

This operation executes the `myjsfile.js` script in a `mongo` shell that connects to the `test` *database* on the `mongod` instance accessible via the `localhost` interface on port `27017`.

Alternately, you can specify the mongodb connection parameters inside of the javascript file using the `Mongo()` constructor. See *Opening New Connections* (page 107) for more information.

You can execute a `.js` file from within the `mongo` shell, using the `load()` function, as in the following:

```
load("myjstest.js")
```

This function loads and executes the `myjstest.js` file.

The `load()` method accepts relative and absolute paths. If the current working directory of the `mongo` shell is `/data/db`, and the `myjstest.js` resides in the `/data/db/scripts` directory, then the following calls within the `mongo` shell would be equivalent:

```
load("scripts/myjstest.js")
load("/data/db/scripts/myjstest.js")
```

Note: There is no search path for the `load()` function. If the desired script is not in the current working directory or the full specified path, `mongo` will not be able to access the file.

2.3.4 Getting Started with the `mongo` Shell

On this page

This document provides a basic introduction to using the `mongo` shell. See `https://docs.mongodb.org/manual/installation` for instructions on installing MongoDB for your system.

Start the `mongo` Shell

To start the `mongo` shell and connect to your `MongoDB` instance running on **localhost** with **default port**:

1. Go to your `<mongodb installation dir>`:

   ```
   cd <mongodb installation dir>
   ```

2. Type `./bin/mongo` to start `mongo`:

```
./bin/mongo
```

If you have added the `<mongodb installation dir>/bin` to the `PATH` environment variable, you can just type `mongo` instead of `./bin/mongo`.

3.To display the database you are using, type `db`:

```
db
```

The operation should return `test`, which is the default database. To switch databases, issue the `use <db>` helper, as in the following example:

```
use <database>
```

To list the available databases, use the helper `show dbs`. See also *mongo-shell-getSiblingDB* to access a different database from the current database without switching your current database context (i.e. `db..`)

To start the `mongo` shell with other options, see *examples of starting up mongo* and `mongo reference` which provides details on the available options.

Note: When starting, `mongo` checks the user's `HOME` directory for a JavaScript file named *.mongorc.js*. If found, `mongo` interprets the content of `.mongorc.js` before displaying the prompt for the first time. If you use the shell to evaluate a JavaScript file or expression, either by using the `--eval` option on the command line or by specifying *a .js file to mongo*, `mongo` will read the `.mongorc.js` file *after* the JavaScript has finished processing. You can prevent `.mongorc.js` from being loaded by using the `--norc` option.

Executing Queries

From the `mongo` shell, you can use the `shell methods` to run queries, as in the following example:

```
db.<collection>.find()
```

- The `db` refers to the current database.
- The `<collection>` is the name of the collection to query. See *Collection Help* (page 114) to list the available collections.

  If the `mongo` shell does not accept the name of the collection, for instance if the name contains a space, hyphen, or starts with a number, you can use an alternate syntax to refer to the collection, as in the following:

  ```
  db["3test"].find()

  db.getCollection("3test").find()
  ```

- The `find()` method is the JavaScript method to retrieve documents from `<collection>`. The `find()` method returns a *cursor* to the results; however, in the `mongo` shell, if the returned cursor is not assigned to a variable using the `var` keyword, then the cursor is automatically iterated up to 20 times to print up to the first 20 documents that match the query. The `mongo` shell will prompt `Type it` to iterate another 20 times.

  You can set the `DBQuery.shellBatchSize` attribute to change the number of iteration from the default value `20`, as in the following example which sets it to `10`:

  ```
  DBQuery.shellBatchSize = 10;
  ```

  For more information and examples on cursor handling in the `mongo` shell, see `https://docs.mongodb.org/manual/core/cursors`.

See also *Cursor Help* (page 114) for list of cursor help in the `mongo` shell.

For more documentation of basic MongoDB operations in the `mongo` shell, see:

- Getting Started with MongoDB[30]
- *mongo Shell Quick Reference* (page 115)
- `https://docs.mongodb.org/manual/core/read-operations`
- `https://docs.mongodb.org/manual/core/write-operations`
- `https://docs.mongodb.org/manual/administration/indexes`

Print

The `mongo` shell automatically prints the results of the `find()` method if the returned cursor is not assigned to a variable using the `var` keyword. To format the result, you can add the `.pretty()` to the operation, as in the following:

```
db.<collection>.find().pretty()
```

In addition, you can use the following explicit print methods in the `mongo` shell:

- `print()` to print without formatting
- `print(tojson(<obj>))` to print with *JSON* formatting and equivalent to `printjson()`
- `printjson()` to print with *JSON* formatting and equivalent to `print(tojson(<obj>))`

Evaluate a JavaScript File

You can execute a `.js` file from within the `mongo` shell, using the `load()` function, as in the following:

```
load("myjstest.js")
```

This function loads and executes the `myjstest.js` file.

The `load()` method accepts relative and absolute paths. If the current working directory of the `mongo` shell is `/data/db`, and the `myjstest.js` resides in the `/data/db/scripts` directory, then the following calls within the `mongo` shell would be equivalent:

```
load("scripts/myjstest.js")
load("/data/db/scripts/myjstest.js")
```

Note: There is no search path for the `load()` function. If the desired script is not in the current working directory or the full specified path, `mongo` will not be able to access the file.

Use a Custom Prompt

You may modify the content of the prompt by creating the variable `prompt` in the shell. The prompt variable can hold strings as well as any arbitrary JavaScript. If `prompt` holds a function that returns a string, `mongo` can display dynamic information in each prompt. Consider the following examples:

Example

[30] http://docs.mongodb.org/getting-started/shell

Create a prompt with the number of operations issued in the current session, define the following variables:

```
cmdCount = 1;
prompt = function() {
             return (cmdCount++) + "> ";
         }
```

The prompt would then resemble the following:

```
1> db.collection.find()
2> show collections
3>
```

Example

To create a `mongo` shell prompt in the form of `<database>@<hostname>$` define the following variables:

```
host = db.serverStatus().host;

prompt = function() {
             return db+"@"+host+"$ ";
         }
```

The prompt would then resemble the following:

```
<database>@<hostname>$ use records
switched to db records
records@<hostname>$
```

Example

To create a `mongo` shell prompt that contains the system up time *and* the number of documents in the current database, define the following prompt variable:

```
prompt = function() {
             return "Uptime:"+db.serverStatus().uptime+" Documents:"+db.stats().objects+" > ";
         }
```

The prompt would then resemble the following:

```
Uptime:5897 Documents:6 > db.people.save({name : "James"});
Uptime:5948 Documents:7 >
```

Use an External Editor in the `mongo` Shell

New in version 2.2.

In the `mongo` shell you can use the `edit` operation to edit a function or variable in an external editor. The `edit` operation uses the value of your environments `EDITOR` variable.

At your system prompt you can define the `EDITOR` variable and start `mongo` with the following two operations:

```
export EDITOR=vim
mongo
```

Then, consider the following example shell session:

```
MongoDB shell version: 2.2.0
> function f() {}
> edit f
> f
function f() {
    print("this really works");
}
> f()
this really works
> o = {}
{ }
> edit o
> o
{ "soDoes" : "this" }
>
```

Note: As `mongo` shell interprets code edited in an external editor, it may modify code in functions, depending on the JavaScript compiler. For `mongo` may convert `1+1` to `2` or remove comments. The actual changes affect only the appearance of the code and will vary based on the version of JavaScript used but will not affect the semantics of the code.

Exit the Shell

To exit the shell, type `quit()` or use the `<Ctrl-c>` shortcut.

See also:

Getting Started Guide[31]

2.3.5 Access the `mongo` Shell Help Information

On this page

In addition to the documentation in the `MongoDB Manual`, the `mongo` shell provides some additional information in its "online" help system. This document provides an overview of accessing this help information.

See also:

- `mongo Manual Page`
- *MongoDB Scripting* (page 101), and
- *mongo Shell Quick Reference* (page 115).

[31] https://docs.mongodb.org/getting-started/shell

Command Line Help

To see the list of options and help for starting the `mongo` shell, use the *--help* option from the command line:

```
mongo --help
```

Shell Help

To see the list of help, in the `mongo` shell, type `help`:

```
help
```

Database Help

- To see the list of databases on the server, use the `show dbs` command:

  ```
  show dbs
  ```

  New in version 2.4: `show databases` is now an alias for `show dbs`

- To see the list of help for methods you can use on the `db` object, call the `db.help()` method:

  ```
  db.help()
  ```

- To see the implementation of a method in the shell, type the `db.<method name>` without the parenthesis (`()`), as in the following example which will return the implementation of the method `db.updateUser()`:

  ```
  db.updateUser
  ```

Collection Help

- To see the list of collections in the current database, use the `show collections` command:

  ```
  show collections
  ```

- To see the help for methods available on the collection objects (e.g. `db.<collection>`), use the `db.<collection>.help()` method:

  ```
  db.collection.help()
  ```

  `<collection>` can be the name of a collection that exists, although you may specify a collection that doesn't exist.

- To see the collection method implementation, type the `db.<collection>.<method>` name without the parenthesis (`()`), as in the following example which will return the implementation of the `save()` method:

  ```
  db.collection.save
  ```

Cursor Help

When you perform *read operations* with the `find()` method in the `mongo` shell, you can use various cursor methods to modify the `find()` behavior and various JavaScript methods to handle the cursor returned from the `find()` method.

- To list the available modifier and cursor handling methods, use the `db.collection.find().help()` command:

  ```
  db.collection.find().help()
  ```

  `<collection>` can be the name of a collection that exists, although you may specify a collection that doesn't exist.

- To see the implementation of the cursor method, type the `db.<collection>.find().<method>` name without the parenthesis (`()`), as in the following example which will return the implementation of the `toArray()` method:

  ```
  db.collection.find().toArray
  ```

Some useful methods for handling cursors are:

- `hasNext()` which checks whether the cursor has more documents to return.
- `next()` which returns the next document and advances the cursor position forward by one.
- `forEach(<function>)` which iterates the whole cursor and applies the `<function>` to each document returned by the cursor. The `<function>` expects a single argument which corresponds to the document from each iteration.

For examples on iterating a cursor and retrieving the documents from the cursor, see `cursor handling`. See also *js-query-cursor-methods* for all available cursor methods.

Type Help

To get a list of the wrapper classes available in the `mongo` shell, such as `BinData()`, type `help misc` in the `mongo` shell:

```
help misc
```

2.3.6 `mongo` Shell Quick Reference

On this page

`mongo` Shell Command History

You can retrieve previous commands issued in the `mongo` shell with the up and down arrow keys. Command history is stored in `~/.dbshell` file. See *.dbshell* for more information.

Command Line Options

The `mongo` shell can be started with numerous options. See `mongo shell` page for details on all available options.

The following table displays some common options for `mongo`:

Option	Description
`--help`	Show command line options
`--nodb`	Start `mongo` shell without connecting to a database. To connect later, see *Opening New Connections* (page 107).
`--shell`	Used in conjunction with a JavaScript file (i.e. <*file.js*>) to continue in the `mongo` shell after running the JavaScript file. See *JavaScript file* (page 109) for an example.

Command Helpers

The `mongo` shell provides various help. The following table displays some common help methods and commands:

Help Methods and Commands	Description
`help`	Show help.
`db.help()`	Show help for database methods.
`db.<collection>.help()`	Show help on collection methods. The `<collection>` can be the name of an existing collection or a non-existing collection.
`show dbs`	Print a list of all databases on the server.
`use <db>`	Switch current database to `<db>`. The `mongo` shell variable `db` is set to the current database.
`show collections`	Print a list of all collections for current database
`show users`	Print a list of users for current database.
`show roles`	Print a list of all roles, both user-defined and built-in, for the current database.
`show profile`	Print the five most recent operations that took 1 millisecond or more. See documentation on the *database profiler* (page 56) for more information.
`show databases`	New in version 2.4: Print a list of all available databases.
`load()`	Execute a JavaScript file. See *Getting Started with the mongo Shell* (page 109) for more information.

Basic Shell JavaScript Operations

The `mongo` shell provides a `JavaScript API` for database operations.

In the `mongo` shell, `db` is the variable that references the current database. The variable is automatically set to the default database `test` or is set when you use the `use <db>` to switch current database.

The following table displays some common JavaScript operations:

JavaScript Database Operations	Description
`db.auth()`	If running in secure mode, authenticate the user.
`coll = db.<collection>`	Set a specific collection in the current database to a variable `coll`, as in the following example: `coll = db.myCollection;` You can perform operations on the `myCollection` using the variable, as in the following example: `coll.find();`
`find()`	Find all documents in the collection and returns a cursor. See the `db.collection.find()` and `https://docs.mongodb.org/manual/tutorial/query-d` for more information and examples. See `https://docs.mongodb.org/manual/core/cursors` for additional information on cursor handling in the `mongo` shell.
`insert()`	Insert a new document into the collection.
`update()`	Update an existing document in the collection. See `https://docs.mongodb.org/manual/core/write-op` for more information.
`save()`	Insert either a new document or update an existing document in the collection. See `https://docs.mongodb.org/manual/core/write-op` for more information.
`remove()`	Delete documents from the collection. See `https://docs.mongodb.org/manual/core/write-op` for more information.
`drop()`	Drops or removes completely the collection.
`createIndex()`	Create a new index on the collection if the index does not exist; otherwise, the operation has no effect.
`db.getSiblingDB()`	Return a reference to another database using this same connection without explicitly switching the current database. This allows for cross database queries. See *mongo-shell-getSiblingDB* for more information.

For more information on performing operations in the shell, see:

- `https://docs.mongodb.org/manual/core/crud`
- `https://docs.mongodb.org/manual/core/read-operations`
- `https://docs.mongodb.org/manual/core/write-operations`
- *js-administrative-methods*

Keyboard Shortcuts

Changed in version 2.2.

The `mongo` shell provides most keyboard shortcuts similar to those found in the `bash` shell or in Emacs. For some functions `mongo` provides multiple key bindings, to accommodate several familiar paradigms.

The following table enumerates the keystrokes supported by the `mongo` shell:

Keystroke	Function
Up-arrow	previous-history
Down-arrow	next-history
Home	beginning-of-line
End	end-of-line
Tab	autocomplete
Left-arrow	backward-character
Right-arrow	forward-character
Ctrl-left-arrow	backward-word
Ctrl-right-arrow	forward-word
Meta-left-arrow	backward-word
Meta-right-arrow	forward-word
Ctrl-A	beginning-of-line
Ctrl-B	backward-char
Ctrl-C	exit-shell
Ctrl-D	delete-char (or exit shell)
Ctrl-E	end-of-line
Ctrl-F	forward-char
Ctrl-G	abort
Ctrl-J	accept-line
Ctrl-K	kill-line
Ctrl-L	clear-screen
Ctrl-M	accept-line
Ctrl-N	next-history
Ctrl-P	previous-history
Ctrl-R	reverse-search-history
Ctrl-S	forward-search-history
Ctrl-T	transpose-chars
Ctrl-U	unix-line-discard
Ctrl-W	unix-word-rubout
Ctrl-Y	yank
Ctrl-Z	Suspend (job control works in linux)
Ctrl-H (i.e. Backspace)	backward-delete-char
Ctrl-I (i.e. Tab)	complete
Meta-B	backward-word
Meta-C	capitalize-word
Meta-D	kill-word
Meta-F	forward-word
Meta-L	downcase-word
Meta-U	upcase-word
Meta-Y	yank-pop
Meta-[Backspace]	backward-kill-word
Meta-<	beginning-of-history
Meta->	end-of-history

Queries

In the `mongo` shell, perform read operations using the `find()` and `findOne()` methods.

The `find()` method returns a cursor object which the `mongo` shell iterates to print documents on screen. By default, `mongo` prints the first 20. The `mongo` shell will prompt the user to "`Type it`" to continue iterating the next 20 results.

The following table provides some common read operations in the `mongo` shell:

Read Operations	Description

`db.collection.find(<query>)`	Find the documents matching the `<query>` criteria in the collection. If the `<query>` criteria is not specified or is empty (i.e `{}`), the read operation selects all documents in the collection.
	The following example selects the documents in the `users` collection with the `name` field equal to `"Joe"`:
	`coll = db.users;`
	`coll.find( { name: "Joe" } );`
	For more information on specifying the `<query>` criteria, see *read-operations-query-argument*.

`db.collection.find(<query>, <projection>)`	Find documents matching the `<query>` criteria and return just specific fields in the `<projection>`.
	The following example selects all documents from the collection but returns only the `name` field and the `_id` field. The `_id` is always returned unless explicitly specified to not return.
	`coll = db.users;`
	`coll.find( { }, { name: true } );`
	For more information on specifying the `<projection>`, see *read-operations-projection*.

`db.collection.find().sort(<sort order>)`	Return results in the specified `<sort order>`.
	The following example selects all documents from the collection and returns the results sorted by the `name` field in ascending order (`1`). Use `-1` for descending order:
	`coll = db.users;`
	`coll.find().sort( { name: 1 } );`

`db.collection.find(<query>).sort(<sort order>)`	Return the documents matching the `<query>` criteria in the specified `<sort order>`.
`db.collection.find( ... ).limit( <n> )`	Limit result to `<n>` rows. Highly recommended if you need only a certain number of rows for best performance.
`db.collection.find( ... ).skip( <n> )`	Skip `<n>` results.
`count()`	Returns total number of documents in the collection.
`db.collection.find(<query>).count()`	Returns the total number of documents that match the query. The `count()` ignores `limit()` and `skip()`. For example, if 100 records match but the limit is 10, `count()` will return 100. This will be faster than iterating yourself, but still take time.

`db.collection.findOne(<query>)`	Find and return a single document. Returns null if not found.
	The following example selects a single document in the `users` collection with the `name` field matches to `"Joe"`:
	`coll = db.users;`
	`coll.findOne( { name: "Joe" } );`
	Internally, the `findOne()` method is the `find()` method with a `limit(1)`.

See `https://docs.mongodb.org/manual/tutorial/query-documents` and `https://docs.mongodb.org/manual/core/read-operations` documentation for more information and examples. See `https://docs.mongodb.org/manual/reference/operator/query` to specify other query operators.

Error Checking Methods

Changed in version 2.6.

The `mongo` shell write methods now integrates the `https://docs.mongodb.org/manual/reference/write-conce` directly into the method execution rather than with a separate `db.getLastError()` method. As such, the write methods now return a `WriteResult()` object that contains the results of the operation, including any write errors and write concern errors.

Previous versions used `db.getLastError()` and `db.getLastErrorObj()` methods to return error information.

Administrative Command Helpers

The following table lists some common methods to support database administration:

JavaScript Database Administration Methods	Description
`db.cloneDatabase(<host>)`	Clone the current database from the `<host>` specified. The `<host>` database instance must be in noauth mode.
`db.copyDatabase(<from>, <to>, <host>)`	Copy the `<from>` database from the `<host>` to the `<to>` database on the current server. The `<host>` database instance must be in `noauth` mode.
`db.fromColl.renameCollection(<toColl>)`	Rename collection from `fromColl` to `<toColl>`.
`db.repairDatabase()`	Repair and compact the current database. This operation can be very slow on large databases.
`db.getCollectionNames()`	Get the list of all collections in the current database.
`db.dropDatabase()`	Drops the current database.

See also *administrative database methods* for a full list of methods.

Opening Additional Connections

You can create new connections within the `mongo` shell.

The following table displays the methods to create the connections:

JavaScript Connection Create Methods	Description
`db = connect("<host><:port>/<dbname>")`	Open a new database connection.

`conn = new Mongo()`
`db = conn.getDB("dbname")`
	Open a connection to a new server using `new Mongo()`.
	Use `getDB()` method of the connection to select a database.

See also *Opening New Connections* (page 107) for more information on the opening new connections from the `mongo` shell.

Miscellaneous

The following table displays some miscellaneous methods:

Method	Description
`Object.bsonsize(<document>)`	Prints the *BSON* size of a <document> in bytes

See the MongoDB JavaScript API Documentation[32] for a full list of JavaScript methods .

Additional Resources

Consider the following reference material that addresses the `mongo` shell and its interface:

- `mongo`
- *js-administrative-methods*
- *database-commands*
- *aggregation-reference*
- Getting Started Guide[33]

Additionally, the MongoDB source code repository includes a jstests directory[34] which contains numerous `mongo` shell scripts.

2.4 MongoDB Tutorials

This page lists the tutorials available as part of the `MongoDB Manual`. In addition to these tutorial in the manual, MongoDB provides *Getting Started Guides* in various driver editions. If there is a process or pattern that you would like to see included here, please open a Jira Case[35].

2.4.1 Installation

- `https://docs.mongodb.org/manual/tutorial/install-mongodb-on-linux`
- `https://docs.mongodb.org/manual/tutorial/install-mongodb-on-red-hat`
- `https://docs.mongodb.org/manual/tutorial/install-mongodb-on-debian`
- `https://docs.mongodb.org/manual/tutorial/install-mongodb-on-ubuntu`
- `https://docs.mongodb.org/manual/tutorial/install-mongodb-on-amazon`
- `https://docs.mongodb.org/manual/tutorial/install-mongodb-on-suse`
- `https://docs.mongodb.org/manual/tutorial/install-mongodb-on-os-x`
- `https://docs.mongodb.org/manual/tutorial/install-mongodb-on-windows`

[32]http://api.mongodb.org/js/index.html
[33]https://docs.mongodb.org/getting-started/shell
[34]https://github.com/mongodb/mongo/tree/master/jstests/
[35]https://jira.mongodb.org/browse/DOCS

2.4.2 Administration

Replica Sets

- *Deploy a Replica Set* (page 160)
- *Convert a Standalone to a Replica Set* (page 172)
- *Add Members to a Replica Set* (page 173)
- *Remove Members from Replica Set* (page 176)
- *Replace a Replica Set Member* (page 178)
- *Adjust Priority for Replica Set Member* (page 179)
- *Resync a Member of a Replica Set* (page 193)
- *Deploy a Geographically Redundant Replica Set* (page 165)
- *Change the Size of the Oplog* (page 187)
- *Force a Member to Become Primary* (page 191)
- *Change Hostnames in a Replica Set* (page 202)
- *Add an Arbiter to Replica Set* (page 171)
- *Convert a Secondary to an Arbiter* (page 185)
- *Configure a Secondary's Sync Target* (page 206)
- *Configure a Delayed Replica Set Member* (page 183)
- *Configure a Hidden Replica Set Member* (page 181)
- *Configure Non-Voting Replica Set Member* (page 184)
- *Prevent Secondary from Becoming Primary* (page 180)
- *Configure Replica Set Tag Sets* (page 194)
- *Manage Chained Replication* (page 201)
- *Reconfigure a Replica Set with Unavailable Members* (page 198)
- *Recover Data after an Unexpected Shutdown* (page 98)
- *Troubleshoot Replica Sets* (page 207)

Sharding

- *Deploy a Sharded Cluster* (page 213)
- *Convert a Replica Set to a Replicated Sharded Cluster* (page 221)
- *Add Shards to a Cluster* (page 219)
- *Remove Shards from an Existing Sharded Cluster* (page 243)
- *Deploy Three Config Servers for Production Deployments* (page 220)
- *Migrate Config Servers with the Same Hostname* (page 230)
- *Migrate Config Servers with Different Hostnames* (page 230)
- *Replace Disabled Config Server* (page 231)

- *Migrate a Sharded Cluster to Different Hardware* (page 232)
- *Backup Cluster Metadata* (page 235)
- *Backup a Small Sharded Cluster with mongodump* (page 88)
- *Backup a Sharded Cluster with Filesystem Snapshots* (page 90)
- *Backup a Sharded Cluster with Database Dumps* (page 92)
- *Restore a Single Shard* (page 95)
- *Restore a Sharded Cluster* (page 96)
- *Schedule Backup Window for Sharded Clusters* (page 94)
- *Manage Shard Tags* (page 254)

Basic Operations

- *Use Database Commands* (page 51)
- *Recover Data after an Unexpected Shutdown* (page 98)
- *Expire Data from Collections by Setting TTL* (page 35)
- *Analyze Performance of Database Operations* (page 56)
- *Rotate Log Files* (page 61)
- `https://docs.mongodb.org/manual/tutorial/roll-back-to-v1.8-index`
- *Manage mongod Processes* (page 52)
- *Back Up and Restore with MongoDB Tools* (page 82)
- *Backup and Restore with Filesystem Snapshots* (page 75)

Security

- `https://docs.mongodb.org/manual/tutorial/configure-linux-iptables-firewall`
- `https://docs.mongodb.org/manual/tutorial/configure-windows-netsh-firewall`
- `https://docs.mongodb.org/manual/tutorial/enable-authentication`
- `https://docs.mongodb.org/manual/tutorial/enable-internal-authentication`
- `https://docs.mongodb.org/manual/tutorial/manage-users-and-roles`
- `https://docs.mongodb.org/manual/tutorial/control-access-to-mongodb-with-kerberos-au`
- `https://docs.mongodb.org/manual/tutorial/create-a-vulnerability-report`

2.4.3 Development Patterns

- `https://docs.mongodb.org/manual/tutorial/perform-two-phase-commits`
- `https://docs.mongodb.org/manual/tutorial/create-an-auto-incrementing-field`
- *Enforce Unique Keys for Sharded Collections* (page 256)
- `https://docs.mongodb.org/manual/applications/aggregation`

- `https://docs.mongodb.org/manual/tutorial/model-data-for-keyword-search`
- `https://docs.mongodb.org/manual/tutorial/limit-number-of-elements-in-updated-array`
- `https://docs.mongodb.org/manual/tutorial/perform-incremental-map-reduce`
- `https://docs.mongodb.org/manual/tutorial/troubleshoot-map-function`
- `https://docs.mongodb.org/manual/tutorial/troubleshoot-reduce-function`
- *Store a JavaScript Function on the Server* (page 65)

2.4.4 Text Search Patterns

- `https://docs.mongodb.org/manual/tutorial/create-text-index-on-multiple-fields`
- `https://docs.mongodb.org/manual/tutorial/specify-language-for-text-index`
- `https://docs.mongodb.org/manual/tutorial/avoid-text-index-name-limit`
- `https://docs.mongodb.org/manual/tutorial/control-results-of-text-search`
- `https://docs.mongodb.org/manual/tutorial/limit-number-of-items-scanned-for-text-sea`

2.4.5 Data Modeling Patterns

- `https://docs.mongodb.org/manual/tutorial/model-embedded-one-to-one-relationships-be`
- `https://docs.mongodb.org/manual/tutorial/model-embedded-one-to-many-relationships-b`
- `https://docs.mongodb.org/manual/tutorial/model-referenced-one-to-many-relationships`
- `https://docs.mongodb.org/manual/tutorial/model-data-for-atomic-operations`
- `https://docs.mongodb.org/manual/tutorial/model-tree-structures-with-parent-referenc`
- `https://docs.mongodb.org/manual/tutorial/model-tree-structures-with-child-reference`
- `https://docs.mongodb.org/manual/tutorial/model-tree-structures-with-materialized-pa`
- `https://docs.mongodb.org/manual/tutorial/model-tree-structures-with-nested-sets`

See also:

The MongoDB Manual contains administrative documentation and tutorials though out several sections. See *Replica Set Tutorials* (page 159) and *Sharded Cluster Tutorials* (page 211) for additional tutorials and information.

CHAPTER 3

Administration Reference

UNIX ulimit Settings **(page 125)** Describes user resources limits (i.e. `ulimit`) and introduces the considerations and optimal configurations for systems that run MongoDB deployments.

System Collections **(page 129)** Introduces the internal collections that MongoDB uses to track per-database metadata, including indexes, collections, and authentication credentials.

Database Profiler Output **(page 130)** Describes the data collected by MongoDB's operation profiler, which introspects operations and reports data for analysis on performance and behavior.

Server Status Output **(page 135)** Provides an example and a high level overview of the output of the `serverStatus` command.

Journaling **(page 147)** Describes the internal operation of MongoDB's journaling facility and outlines how the journal allows MongoDB to provide provides durability and crash resiliency.

Exit Codes and Statuses **(page 150)** Lists the unique codes returned by `mongos` and `mongod` processes upon exit.

3.1 UNIX `ulimit` Settings

On this page

Most UNIX-like operating systems, including Linux and OS X, provide ways to limit and control the usage of system resources such as threads, files, and network connections on a per-process and per-user basis. These "ulimits" prevent single users from using too many system resources. Sometimes, these limits have low default values that can cause a number of issues in the course of normal MongoDB operation.

Note: Red Hat Enterprise Linux and CentOS 6 place a max process limitation of 1024 which overrides `ulimit` settings. Create a file named `/etc/security/limits.d/99-mongodb-nproc.conf` with new `soft nproc` and `hard nproc` values to increase the process limit. See `/etc/security/limits.d/90-nproc.conf` file as an example.

3.1.1 Resource Utilization

`mongod` and `mongos` each use threads and file descriptors to track connections and manage internal operations. This section outlines the general resource utilization patterns for MongoDB. Use these figures in combination with the actual information about your deployment and its use to determine ideal `ulimit` settings.

Generally, all `mongod` and `mongos` instances:

- track each incoming connection with a file descriptor *and* a thread.
- track each internal thread or *pthread* as a system process.

mongod

- 1 file descriptor for each data file in use by the `mongod` instance.
- 1 file descriptor for each journal file used by the `mongod` instance when `storage.journal.enabled` is `true`.
- In replica sets, each `mongod` maintains a connection to all other members of the set.

`mongod` uses background threads for a number of internal processes, including *TTL collections* (page 35), replication, and replica set health checks, which may require a small number of additional resources.

mongos

In addition to the threads and file descriptors for client connections, `mongos` must maintain connects to all config servers and all shards, which includes all members of all replica sets.

For `mongos`, consider the following behaviors:

- `mongos` instances maintain a connection pool to each shard so that the `mongos` can reuse connections and quickly fulfill requests without needing to create new connections.
- You can limit the number of incoming connections using the `maxIncomingConnections` run-time option. By restricting the number of incoming connections you can prevent a cascade effect where the `mongos` creates too many connections on the `mongod` instances.

Note: Changed in version 2.6: MongoDB removed the upward limit on the `maxIncomingConnections` setting.

3.1.2 Review and Set Resource Limits

ulimit

You can use the `ulimit` command at the system prompt to check system limits, as in the following example:

```
$ ulimit -a
-t: cpu time (seconds)          unlimited
-f: file size (blocks)          unlimited
-d: data seg size (kbytes)      unlimited
-s: stack size (kbytes)         8192
-c: core file size (blocks)     0
-m: resident set size (kbytes) unlimited
-u: processes                   192276
-n: file descriptors            21000
```

```
-l: locked-in-memory size (kb) 40000
-v: address space (kb)         unlimited
-x: file locks                 unlimited
-i: pending signals            192276
-q: bytes in POSIX msg queues  819200
-e: max nice                   30
-r: max rt priority            65
-N 15:                         unlimited
```

`ulimit` refers to the per-*user* limitations for various resources. Therefore, if your `mongod` instance executes as a user that is also running multiple processes, or multiple `mongod` processes, you might see contention for these resources. Also, be aware that the `processes` value (i.e. `-u`) refers to the combined number of distinct processes and sub-process threads.

You can change `ulimit` settings by issuing a command in the following form:

```
ulimit -n <value>
```

There are both "hard" and the "soft" `ulimit`s that affect MongoDB's performance. The "hard" `ulimit` refers to the maximum number of processes that a user can have active at any time. This is the ceiling: no non-root process can increase the "hard" `ulimit`. In contrast, the "soft" `ulimit` is the limit that is actually enforced for a session or process, but any process can increase it up to "hard" `ulimit` maximum.

A low "soft" `ulimit` can cause `can't create new thread, closing connection` errors if the number of connections grows too high. For this reason, it is extremely important to set *both* `ulimit` values to the recommended values.

`ulimit` will modify both "hard" and "soft" values unless the `-H` or `-S` modifiers are specified when modifying limit values.

For many distributions of Linux you can change values by substituting the `-n` option for any possible value in the output of `ulimit -a`. On OS X, use the `launchctl limit` command. See your operating system documentation for the precise procedure for changing system limits on running systems.

After changing the `ulimit` settings, you *must* restart the process to take advantage of the modified settings. You can use the `/proc` file system to see the current limitations on a running process.

Depending on your system's configuration, and default settings, any change to system limits made using `ulimit` may revert following system a system restart. Check your distribution and operating system documentation for more information.

Note: SUSE Linux Enterprise Server 11 and potentially other versions of SLES and other SUSE distributions ship with virtual memory address space limited to 8GB by default. This *must* be adjusted in order to prevent virtual memory allocation failures as the database grows.

The SLES packages for MongoDB adjust these limits in the default scripts, but you will need to make this change manually if you are using custom scripts and/or the tarball release rather than the SLES packages.

Recommended `ulimit` Settings

Every deployment may have unique requirements and settings; however, the following thresholds and settings are particularly important for `mongod` and `mongos` deployments:

- `-f` (file size): `unlimited`
- `-t` (cpu time): `unlimited`

- `-v` (virtual memory): `unlimited` [1]
- `-n` (open files): `64000`
- `-m` (memory size): `unlimited` [1] [2]
- `-u` (processes/threads): `64000`

Always remember to restart your `mongod` and `mongos` instances after changing the `ulimit` settings to ensure that the changes take effect.

Linux distributions using Upstart

For Linux distributions that use Upstart, you can specify limits within service scripts if you start `mongod` and/or `mongos` instances as Upstart services. You can do this by using `limit` stanzas[3].

Specify the *Recommended ulimit Settings* (page 127), as in the following example:

```
limit fsize unlimited unlimited  # (file size)
limit cpu unlimited unlimited    # (cpu time)
limit as unlimited unlimited     # (virtual memory size)
limit nofile 64000 64000         # (open files)
limit nproc 64000 64000          # (processes/threads)
```

Each `limit` stanza sets the "soft" limit to the first value specified and the "hard" limit to the second.

After changing `limit` stanzas, ensure that the changes take effect by restarting the application services, using the following form:

```
restart <service name>
```

Linux distributions using `systemd`

For Linux distributions that use `systemd`, you can specify limits within the `[Service]` sections of service scripts if you start `mongod` and/or `mongos` instances as `systemd` services. You can do this by using resource limit directives[4].

Specify the *Recommended ulimit Settings* (page 127), as in the following example:

```
[Service]
# Other directives omitted
# (file size)
LimitFSIZE=infinity
# (cpu time)
LimitCPU=infinity
# (virtual memory size)
LimitAS=infinity
# (open files)
LimitNOFILE=64000
# (processes/threads)
LimitNPROC=64000
```

Each `systemd` limit directive sets both the "hard" and "soft" limits to the value specified.

[1] If you limit virtual or resident memory size on a system running MongoDB the operating system will refuse to honor additional allocation requests.

[2] The `-m` parameter to `ulimit` has no effect on Linux systems with kernel versions more recent than 2.4.30. You may omit `-m` if you wish.

[3] http://upstart.ubuntu.com/wiki/Stanzas#limit

[4] http://www.freedesktop.org/software/systemd/man/systemd.exec.html#LimitCPU=

After changing `limit` stanzas, ensure that the changes take effect by restarting the application services, using the following form:

```
systemctl restart <service name>
```

`/proc` File System

Note: This section applies only to Linux operating systems.

The `/proc` file-system stores the per-process limits in the file system object located at `/proc/<pid>/limits`, where `<pid>` is the process's *PID* or process identifier. You can use the following `bash` function to return the content of the `limits` object for a process or processes with a given name:

```
return-limits(){

     for process in $@; do
          process_pids=`ps -C $process -o pid --no-headers | cut -d " " -f 2`

          if [ -z $@ ]; then
             echo "[no $process running]"
          else
             for pid in $process_pids; do
                   echo "[$process #$pid -- limits]"
                   cat /proc/$pid/limits
             done
          fi

     done

}
```

You can copy and paste this function into a current shell session or load it as part of a script. Call the function with one the following invocations:

```
return-limits mongod
return-limits mongos
return-limits mongod mongos
```

3.2 System Collections

On this page

3.2.1 Synopsis

MongoDB stores system information in collections that use the `<database>.system.*` *namespace*, which MongoDB reserves for internal use. Do not create collections that begin with `system`.

MongoDB also stores some additional instance-local metadata in the `local database`, specifically for replication purposes.

3.2.2 Collections

System collections include these collections stored in the `admin` database:

`admin.system.`**`roles`**
New in version 2.6.

The `admin.system.roles` (page 130) collection stores custom roles that administrators create and assign to users to provide access to specific resources.

`admin.system.`**`users`**
Changed in version 2.6.

The `admin.system.users` (page 130) collection stores the user's authentication credentials as well as any roles assigned to the user. Users may define authorization roles in the `admin.system.roles` (page 130) collection.

`admin.system.`**`version`**
New in version 2.6.

Stores the schema version of the user credential documents.

System collections also include these collections stored directly in each database:

`<database>.system.`**`namespaces`**
Deprecated since version 3.0: Access this data using `listCollections`.

The `<database>.system.namespaces` (page 130) collection contains information about all of the database's collections.

`<database>.system.`**`indexes`**
Deprecated since version 3.0: Access this data using `listIndexes`.

The `<database>.system.indexes` (page 130) collection lists all the indexes in the database.

`<database>.system.`**`profile`**
The `<database>.system.profile` (page 130) collection stores database profiling information. For information on profiling, see *Database Profiling* (page 39).

`<database>.system.`**`js`**
The `<database>.system.js` (page 130) collection holds special JavaScript code for use in *server side JavaScript* (page 102). See *Store a JavaScript Function on the Server* (page 65) for more information.

3.3 Database Profiler Output

On this page

The database profiler captures data information about read and write operations, cursor operations, and database commands. To configure the database profile and set the thresholds for capturing profile data, see the *Analyze Performance of Database Operations* (page 56) section.

The database profiler writes data in the `system.profile` (page 130) collection, which is a *capped collection*. To view the profiler's output, use normal MongoDB queries on the `system.profile` (page 130) collection.

Note: Because the database profiler writes data to the `system.profile` (page 130) collection in a database, the profiler will profile some write activity, even for databases that are otherwise read-only.

3.3.1 Example `system.profile` Document

The documents in the `system.profile` (page 130) collection have the following form. This example document reflects an insert operation:

```
{
    "op" : "insert",
    "ns" : "test.orders",
    "query" : {
       "_id" : 1,
       "cust_id" : "A123",
       "amount" : 500,
       "status" : "A"
    },
    "ninserted" : 1,
    "keyUpdates" : 0,
    "writeConflicts" : 0,
    "numYield" : 0,
    "locks" : {
          "Global" : {
             "acquireCount" : {
                "w" : NumberLong(1)
             }
          },
          "MMAPV1Journal" : {
             "acquireCount" : {
                "w" : NumberLong(2)
             }
          },
          "Database" : {
             "acquireCount" : {
                "w" : NumberLong(1)
             }
          },
          "Collection" : {
             "acquireCount" : {
                "W" : NumberLong(1)
             }
          }
       },
    ,
    "millis" : 0,
    "execStats" : {
    },
    "ts" : ISODate("2012-12-10T19:31:28.977Z"),
    "client" : "127.0.0.1",
    "allUsers" : [ ],
    "user" : ""
}
```

3.3.2 Output Reference

For any single operation, the documents created by the database profiler will include a subset of the following fields. The precise selection of fields in these documents depends on the type of operation.

Note: For the output specific to the version of your MongoDB, refer to the appropriate version of the MongoDB Manual.

`system.profile.`**`op`**

The type of operation. The possible values are:

- `insert`
- `query`
- `update`
- `remove`
- `getmore`
- `command`

`system.profile.`**`ns`**

The *namespace* the operation targets. Namespaces in MongoDB take the form of the *database*, followed by a dot (`.`), followed by the name of the *collection*.

`system.profile.`**`query`**

The *query document* used, or for an insert operation, the inserted document. If the document exceeds 50 kilobytes, the value is a string summary of the object. If the string summary exceeds 50 kilobytes, the string summary is truncated, denoted with an ellipsis (`...`) at the end of the string.

Changed in version 3.0.4: For `"getmore"` (page 132) operations on cursors returned from a `db.collection.find()` or a `db.collection.aggregate()`, the `query` (page 132) field contains respectively the query predicate or the issued `aggregate` command document. For details on the `aggregate` command document, see the `aggregate` reference page.

`system.profile.`**`command`**

The command operation. If the command document exceeds 50 kilobytes, the value is a string summary of the object. If the string summary exceeds 50 kilobytes, the string summary is truncated, denoted with an ellipsis (`...`) at the end of the string.

`system.profile.`**`updateobj`**

The `<update>` document passed in during an `update` operation. If the document exceeds 50 kilobytes, the value is a string summary of the object. If the string summary exceeds 50 kilobytes, the string summary is truncated, denoted with an ellipsis (`...`) at the end of the string.

`system.profile.`**`cursorid`**

The ID of the cursor accessed by a `query` and `getmore` operations.

`system.profile.`**`ntoreturn`**

The number of documents the operation specified to return. For example, the `profile` command would return one document (a results document) so the `ntoreturn` (page 132) value would be `1`. The `limit(5)` command would return five documents so the `ntoreturn` (page 132) value would be `5`.

If the `ntoreturn` (page 132) value is `0`, the command did not specify a number of documents to return, as would be the case with a simple `find()` command with no limit specified.

`system.profile.`**`ntoskip`**

The number of documents the `skip()` method specified to skip.

`system.profile.`**`nscanned`**
The number of documents that MongoDB scans in the `index` in order to carry out the operation.

In general, if `nscanned` (page 132) is much higher than `nreturned` (page 134), the database is scanning many objects to find the target objects. Consider creating an index to improve this.

`system.profile.`**`nscannedObjects`**
The number of documents that MongoDB scans from the collection in order to carry out the operation.

`system.profile.`**`moved`**
Changed in version 3.0.0: Only appears when using the MMAPv1 storage engine.

This field appears with a value of `true` when an update operation moved one or more documents to a new location on disk. If the operation did not result in a move, this field does not appear. Operations that result in a move take more time than in-place updates and typically occur as a result of document growth.

`system.profile.`**`nmoved`**
Changed in version 3.0.0: Only appears when using the MMAPv1 storage engine.

The number of documents the operation moved on disk. This field appears only if the operation resulted in a move. The field's implicit value is zero, and the field is present only when non-zero.

`system.profile.`**`scanAndOrder`**
`scanAndOrder` (page 133) is a boolean that is `true` when a query **cannot** use the ordering in the index to return the requested sorted results; i.e. MongoDB must sort the documents after it receives the documents from a cursor. The field only appears when the value is `true`.

`system.profile.`**`ndeleted`**
The number of documents deleted by the operation.

`system.profile.`**`ninserted`**
The number of documents inserted by the operation.

`system.profile.`**`nMatched`**
New in version 2.6.

The number of documents that match the `system.profile.query` (page 132) condition for the update operation.

`system.profile.`**`nModified`**
New in version 2.6.

The number of documents modified by the update operation.

`system.profile.`**`upsert`**
A boolean that indicates the update operation's `upsert` option value. Only appears if `upsert` is true.

`system.profile.`**`keyUpdates`**
The number of `index` keys the update changed in the operation. Changing an index key carries a small performance cost because the database must remove the old key and inserts a new key into the B-tree index.

`system.profile.`**`writeConflicts`**
New in version 3.0.0.

The number of conflicts encountered during the write operation; e.g. an `update` operation attempts to modify the same document as another `update` operation. See also *write conflict*.

`system.profile.`**`numYield`**
The number of times the operation yielded to allow other operations to complete. Typically, operations yield when they need access to data that MongoDB has not yet fully read into memory. This allows other operations that have data in memory to complete while MongoDB reads in data for the yielding operation. For more information, see *the FAQ on when operations yield*.

`system.profile.`**`locks`**

New in version 3.0.0: `locks` (page 133) replaces the `lockStats` field.

The `system.profile.locks` (page 133) provides information for various *lock types and lock modes* held during the operation.

The possible lock types are:

- `Global` represents global lock.
- `MMAPV1Journal` represents MMAPv1 storage engine specific lock to synchronize journal writes; for non-MMAPv1 storage engines, the mode for `MMAPV1Journal` is empty.
- `Database` represents database lock.
- `Collection` represents collection lock.
- `Metadata` represents metadata lock.
- `oplog` represents lock on the *oplog*.

The possible locking modes for the lock types are as follows:

- `R` represents Shared (S) lock.
- `W` represents Exclusive (X) lock.
- `r` represents Intent Shared (IS) lock.
- `w` represents Intent Exclusive (IX) lock.

The returned lock information for the various lock types include:

`system.profile.locks.`**`acquireCount`**

Number of times the operation acquired the lock in the specified mode.

`system.profile.locks.`**`acquireWaitCount`**

Number of times the operation had to wait for the `acquireCount` (page 134) lock acquisitions because the locks were held in a conflicting mode. `acquireWaitCount` (page 134) is less than or equal to `acquireCount` (page 134).

`system.profile.locks.`**`timeAcquiringMicros`**

Cumulative time in microseconds that the operation had to wait to acquire the locks.

`timeAcquiringMicros` (page 134) divided by `acquireWaitCount` (page 134) gives an approximate average wait time for the particular lock mode.

`system.profile.locks.`**`deadlockCount`**

Number of times the operation encountered deadlocks while waiting for lock acquisitions.

For more information on lock modes, see *faq-concurrency-locking*.

`system.profile.`**`nreturned`**

The number of documents returned by the operation.

`system.profile.`**`responseLength`**

The length in bytes of the operation's result document. A large `responseLength` (page 134) can affect performance. To limit the size of the result document for a query operation, you can use any of the following:

- *Projections*
- `The limit() method`
- `The batchSize() method`

Note: When MongoDB writes query profile information to the log, the `responseLength` (page 134) value is in a field named `reslen`.

`system.profile.`**`millis`**
The time in milliseconds from the perspective of the `mongod` from the beginning of the operation to the end of the operation.

`system.profile.`**`execStats`**
Changed in version 3.0.

A document that contains the execution statistics of the query operation. For other operations, the value is an empty document.

The `system.profile.execStats` (page 135) presents the statistics as a tree; each node provides the statistics for the operation executed during that stage of the query operation.

Note: The following fields list for `execStats` (page 135) is not meant to be exhaustive as the returned fields vary per stage.

`system.profile.execStats.`**`stage`**
New in version 3.0: `stage` (page 135) replaces the `type` field.

The descriptive name for the operation performed as part of the query execution; e.g.

- `COLLSCAN` for a collection scan
- `IXSCAN` for scanning index keys
- `FETCH` for retrieving documents

`system.profile.execStats.`**`inputStages`**
New in version 3.0: `inputStages` (page 135) replaces the `children` field.

An array that contains statistics for the operations that are the input stages of the current stage.

`system.profile.`**`ts`**
The timestamp of the operation.

`system.profile.`**`client`**
The IP address or hostname of the client connection where the operation originates.

For some operations, such as `db.eval()`, the client is `0.0.0.0:0` instead of an actual client.

`system.profile.`**`allUsers`**
An array of authenticated user information (user name and database) for the session. See also `https://docs.mongodb.org/manual/core/security-users`.

`system.profile.`**`user`**
The authenticated user who ran the operation. If the operation was not run by an authenticated user, this field's value is an empty string.

3.4 Server Status Output

This document provides a quick overview and example of the `serverStatus` command. The helper `db.serverStatus()` in the `mongo` shell provides access to this output. For full documentation of the content of this output, see `https://docs.mongodb.org/manual/reference/command/serverStatus`.

Note: The output fields vary depending on the version of MongoDB, underlying operating system platform, the storage engine, and the kind of node, including `mongos`, `mongod` or *replica set* member. For the serverStatus output specific to the version of your MongoDB, refer to the appropriate version of the MongoDB Manual.

Changed in version 3.0: The server status output no longer includes the workingSet, indexCounters, and recordStats sections. The *server-status-instance-information* section displays information regarding the specific mongod and mongos and its state.

```
"host" : "<hostname>",
"version" : "<version>",
"process" : "<mongod|mongos>",
"pid" : <num>,
"uptime" : <num>,
"uptimeMillis" : <num>,
"uptimeEstimate" : <num>,
"localTime" : ISODate(""),
```

The *server-status-asserts* document reports the number of assertions or errors produced by the server:

```
"asserts" : {
   "regular" : <num>,
   "warning" : <num>,
   "msg" : <num>,
   "user" : <num>,
   "rollovers" : <num>
},
```

The *server-status-backgroundflushing* document reports on the process MongoDB uses to write data to disk. The *server-status-backgroundflushing* information only returns for instances that use the MMAPv1 storage engine:

```
"backgroundFlushing" : {
   "flushes" : <num>,
   "total_ms" : <num>,
   "average_ms" : <num>,
   "last_ms" : <num>,
   "last_finished" : ISODate("")
},
```

The *server-status-connections* field reports on MongoDB's current number of open incoming connections:

New in version 2.4: The totalCreated field.

```
"connections" : {
   "current" : <num>,
   "available" : <num>,
   "totalCreated" : NumberLong(<num>)
},
```

The *server-status-cursors* document reports on current cursor use and state:

```
"cursors" : {
   "note" : "deprecated, use server status metrics",
   "clientCursors_size" : <num>,
   "totalOpen" : <num>,
   "pinned" : <num>,
   "totalNoTimeout" : <num>,
   "timedOut" : <num>
},
```

The *server-status-journaling* document reports on data that reflect this mongod instance's journaling-related operations and performance during a *journal group commit interval* (page 65). The *server-status-journaling* information only returns for instances that use the MMAPv1 storage engine and have journaling enabled:

```
"dur" : {
   "commits" : <num>,
   "journaledMB" : <num>,
   "writeToDataFilesMB" : <num>,
   "compression" : <num>,
   "commitsInWriteLock" : <num>,
   "earlyCommits" : <num>,
   "timeMs" : {
      "dt" : <num>,
      "prepLogBuffer" : <num>,
      "writeToJournal" : <num>,
      "writeToDataFiles" : <num>,
      "remapPrivateView" : <num>,
      "commits" : <num>,
      "commitsInWriteLock" : <num>
   }
},
```

The fields in the *server-status-extra-info* document provide platform specific information. The following example block is from a Linux-based system:

```
"extra_info" : {
   "note" : "fields vary by platform",
   "heap_usage_bytes" : <num>,
   "page_faults" : <num>
},
```

The *server-status-globallock* field reports on MongoDB's global system lock. In most cases the *locks* document provides more fine grained data that reflects lock use:

```
"globalLock" : {
   "totalTime" : <num>,
   "currentQueue" : {
      "total" : <num>,
      "readers" : <num>,
      "writers" : <num>
   },
   "activeClients" : {
      "total" : <num>,
      "readers" : <num>,
      "writers" : <num>
   }
},
```

The *server-status-locks* section reports statistics for each lock type and mode:

```
"locks" : {
   "Global" : {
         "acquireCount" : {
            "r" : NumberLong(<num>),
            "w" : NumberLong(<num>),
            "R" : NumberLong(<num>),
            "W" : NumberLong(<num>)
         },
         "acquireWaitCount" : {
            "r" : NumberLong(<num>),
            "w" : NumberLong(<num>),
            "R" : NumberLong(<num>),
            "W" : NumberLong(<num>)
         },
         "timeAcquiringMicros" : {
            "r" : NumberLong(<num>),
            "w" : NumberLong(<num>),
            "R" : NumberLong(<num>),
            "W" : NumberLong(<num>)
         },
         "deadlockCount" : {
            "r" : NumberLong(<num>),
            "w" : NumberLong(<num>),
            "R" : NumberLong(<num>),
            "W" : NumberLong(<num>)
         }
   },
   "MMAPV1Journal" : {
         "acquireCount" : {
            "r" : NumberLong(<num>),
            "w" : NumberLong(<num>),
            "R" : NumberLong(<num>),
            "W" : NumberLong(<num>)
         },
         "acquireWaitCount" : {
            "r" : NumberLong(<num>),
            "w" : NumberLong(<num>),
            "R" : NumberLong(<num>),
            "W" : NumberLong(<num>)
         },
         "timeAcquiringMicros" : {
            "r" : NumberLong(<num>),
            "w" : NumberLong(<num>),
            "R" : NumberLong(<num>),
            "W" : NumberLong(<num>)
         },
         "deadlockCount" : {
            "r" : NumberLong(<num>),
            "w" : NumberLong(<num>),
            "R" : NumberLong(<num>),
            "W" : NumberLong(<num>)
         }
   },
   "Database" : {
         "acquireCount" : {
            "r" : NumberLong(<num>),
            "w" : NumberLong(<num>),
            "R" : NumberLong(<num>),
            "W" : NumberLong(<num>)
         },
         "acquireWaitCount" : {
            "r" : NumberLong(<num>),
            "w" : NumberLong(<num>),
            "R" : NumberLong(<num>),
            "W" : NumberLong(<num>)
         },
         "timeAcquiringMicros" : {
            "r" : NumberLong(<num>),
            "w" : NumberLong(<num>),
            "R" : NumberLong(<num>),
                "W" : NumberLong(<num>)
            },
            "deadlockCount" : {
                "r" : NumberLong(<num>),
                "w" : NumberLong(<num>),
                "R" : NumberLong(<num>),
                "W" : NumberLong(<num>)
            }
    },
    "Collection" : {
            "acquireCount" : {
                "r" : NumberLong(<num>),
                "w" : NumberLong(<num>),
                "R" : NumberLong(<num>),
                "W" : NumberLong(<num>)
            },
            "acquireWaitCount" : {
                "r" : NumberLong(<num>),
                "w" : NumberLong(<num>),
                "R" : NumberLong(<num>),
                "W" : NumberLong(<num>)
            },
            "timeAcquiringMicros" : {
                "r" : NumberLong(<num>),
                "w" : NumberLong(<num>),
                "R" : NumberLong(<num>),
                "W" : NumberLong(<num>)
            },
            "deadlockCount" : {
                "r" : NumberLong(<num>),
                "w" : NumberLong(<num>),
                "R" : NumberLong(<num>),
                "W" : NumberLong(<num>)
            }
    },
    "Metadata" : {
            "acquireCount" : {
                "r" : NumberLong(<num>),
                "w" : NumberLong(<num>),
                "R" : NumberLong(<num>),
                "W" : NumberLong(<num>)
            },
            "acquireWaitCount" : {
                "r" : NumberLong(<num>),
                "w" : NumberLong(<num>),
                "R" : NumberLong(<num>),
                "W" : NumberLong(<num>)
            },
            "timeAcquiringMicros" : {
                "r" : NumberLong(<num>),
                "w" : NumberLong(<num>),
                "R" : NumberLong(<num>),
                "W" : NumberLong(<num>)
            },
            "deadlockCount" : {
                "r" : NumberLong(<num>),
                "w" : NumberLong(<num>),
                "R" : NumberLong(<num>),
                 "W" : NumberLong(<num>)
              }
        },
        "oplog" : {
              "acquireCount" : {
                 "r" : NumberLong(<num>),
                 "w" : NumberLong(<num>),
                 "R" : NumberLong(<num>),
                 "W" : NumberLong(<num>)
              },
              "acquireWaitCount" : {
                 "r" : NumberLong(<num>),
                 "w" : NumberLong(<num>),
                 "R" : NumberLong(<num>),
                 "W" : NumberLong(<num>)
              },
              "timeAcquiringMicros" : {
                 "r" : NumberLong(<num>),
                 "w" : NumberLong(<num>),
                 "R" : NumberLong(<num>),
                 "W" : NumberLong(<num>)
              },
              "deadlockCount" : {
                 "r" : NumberLong(<num>),
                 "w" : NumberLong(<num>),
                 "R" : NumberLong(<num>),
                 "W" : NumberLong(<num>)
              }
        }
     },
```

The *server-status-network* document reports on network use and state:

```
"network" : {
   "bytesIn" : <num>,
   "bytesOut" : <num>,
   "numRequests" : <num>
},
```

The *server-status-opcounters* document reports the number of operations this MongoDB instance has processed:

```
"opcounters" : {
   "insert" : <num>,
   "query" : <num>,
   "update" : <num>,
   "delete" : <num>,
   "getmore" : <num>,
   "command" : <num>
},
```

The *server-status-opcounters-repl* document reports the number of replicated operations:

```
"opcountersRepl" : {
   "insert" : <num>,
   "query" : <num>,
   "update" : <num>,
   "delete" : <num>,
   "getmore" : <num>,
   "command" : <num>
},
```

The *server-status-storage-engine* document reports details about the current storage engine:

```
"storageEngine" : {
   "name" : <string>
},
```

The *server-status-writebacksqueued* document reports the number of *writebacks*:

```
"writeBacksQueued" : <num>,
```

The *server-status-memory* field reports on MongoDB's current memory use:

```
"mem" : {
   "bits" : <num>,
   "resident" : <num>,
   "virtual" : <num>,
   "supported" : <boolean>,
   "mapped" : <num>,
   "mappedWithJournal" : <num>,
   "note" : "not all mem info support on this platform"
},
```

The *server-status-repl* document reports on the state of replication and the *replica set*. This document only appears for replica sets.

```
"repl" : {
   "setName" : <string>,
   "setVersion" : <num>,
   "ismaster" : <boolean>,
   "secondary" : <boolean>,
   "hosts" : [
         <hostname>,
         <hostname>,
         <hostname>
   ],
   "primary" : <hostname>,
   "me" : <hostname>,
   "electionId" : ObjectId(""),
   "rbid" : <num>,
   "slaves" : [
         {
             "rid" : <ObjectId>,
             "optime" : <timestamp>,
             "host" : <hostname>,
             "memberID" : <num>
         }
   ],
},
```

The *server-status-range-deleter* document reports the number of operations this MongoDB instance has processed. The `rangeDeleter` document is only present in the output of `serverStatus` when explicitly enabled.

```
"rangeDeleter" : {
   "lastDeleteStats" : [
      {
        "deletedDocs" : NumberLong(<num>),
        "queueStart" : <date>,
        "queueEnd" : <date>,
        "deleteStart" : <date>,
        "deleteEnd" : <date>,
        "waitForReplStart" : <date>,
        "waitForReplEnd" : <date>
      }
   ]
}
```

The *server-status-security* document reports details about the security features and use:

```
"security" : {
   "SSLServerSubjectName": <string>,
   "SSLServerHasCertificateAuthority": <boolean>,
   "SSLServerCertificateExpirationDate": <date>
},
```

The *server-status-metrics* document contains a number of operational metrics that are useful for monitoring the state and workload of a `mongod` instance.

New in version 2.4.

Changed in version 2.6: Added the `cursor` document.

```
"metrics" : {
   "command": {
         "<command>": {
            "failed": <num>,
            "total": <num>
         }
   },
   "cursor" : {
         "timedOut" : NumberLong(<num>),
         "open" : {
            "noTimeout" : NumberLong(<num>),
            "pinned" : NumberLong(<num>),
            "multiTarget" : NumberLong(<num>),
            "singleTarget" : NumberLong(<num>),
            "total" : NumberLong(<num>),
         }
   },
   "document" : {
         "deleted" : NumberLong(<num>),
         "inserted" : NumberLong(<num>),
         "returned" : NumberLong(<num>),
         "updated" : NumberLong(<num>)
   },
   "getLastError" : {
         "wtime" : {
            "num" : <num>,
            "totalMillis" : <num>
         },
         "wtimeouts" : NumberLong(<num>)
   },
   "operation" : {
         "fastmod" : NumberLong(<num>),
         "idhack" : NumberLong(<num>),
         "scanAndOrder" : NumberLong(<num>)
   },
   "queryExecutor": {
```

```
            "scanned" : NumberLong(<num>)
      },
      "record" : {
            "moves" : NumberLong(<num>)
      },
      "repl" : {
            "apply" : {
               "batches" : {
                  "num" : <num>,
                  "totalMillis" : <num>
               },
               "ops" : NumberLong(<num>)
            },
            "buffer" : {
               "count" : NumberLong(<num>),
               "maxSizeBytes" : <num>,
               "sizeBytes" : NumberLong(<num>)
            },
            "network" : {
               "bytes" : NumberLong(<num>),
               "getmores" : {
                  "num" : <num>,
                  "totalMillis" : <num>
               },
               "ops" : NumberLong(<num>),
               "readersCreated" : NumberLong(<num>)
            },
            "oplog" : {
               "insert" : {
                  "num" : <num>,
                  "totalMillis" : <num>
               },
               "insertBytes" : NumberLong(<num>)
            },
            "preload" : {
               "docs" : {
                  "num" : <num>,
                  "totalMillis" : <num>
               },
               "indexes" : {
                  "num" : <num>,
                  "totalMillis" : <num>
               }
            }
      },
      "storage" : {
            "freelist" : {
               "search" : {
                  "bucketExhausted" : <num>,
                  "requests" : <num>,
                  "scanned" : <num>
               }
            }
      },
      "ttl" : {
            "deletedDocuments" : NumberLong(<num>),
            "passes" : NumberLong(<num>)
      }
```

```
},
```

The *server-status-wiredTiger* statistics section reports details about the WiredTiger statistics:

New in version 3.0: *server-status-wiredTiger* statistics section. This section appears only for the WiredTiger storage engine.

```
"wiredTiger" : {
   "uri" : "statistics:",
   "LSM" : {
         "sleep for LSM checkpoint throttle" : <num>,
         "sleep for LSM merge throttle" : <num>,
         "rows merged in an LSM tree" : <num>,
         "application work units currently queued" : <num>,
         "merge work units currently queued" : <num>,
         "tree queue hit maximum" : <num>,
         "switch work units currently queued" : <num>,
         "tree maintenance operations scheduled" : <num>,
         "tree maintenance operations discarded" : <num>,
         "tree maintenance operations executed" : <num>
   },
   "async" : {
         "number of allocation state races" : <num>,
         "number of operation slots viewed for allocation" : <num>,
         "current work queue length" : <num>,
         "number of flush calls" : <num>,
         "number of times operation allocation failed" : <num>,
         "maximum work queue length" : <num>,
         "number of times worker found no work" : <num>,
         "total allocations" : <num>,
         "total compact calls" : <num>,
         "total insert calls" : <num>,
         "total remove calls" : <num>,
         "total search calls" : <num>,
         "total update calls" : <num>
   },
   "block-manager" : {
         "mapped bytes read" : <num>,
         "bytes read" : <num>,
         "bytes written" : <num>,
         "mapped blocks read" : <num>,
         "blocks pre-loaded" : <num>,
         "blocks read" : <num>,
         "blocks written" : <num>
   },
   "cache" : {
         "tracked dirty bytes in the cache" : <num>,
         "bytes currently in the cache" : <num>,
         "maximum bytes configured" : <num>,
         "bytes read into cache" : <num>,
         "bytes written from cache" : <num>,
         "pages evicted by application threads" : <num>,
         "checkpoint blocked page eviction" : <num>,
         "unmodified pages evicted" : <num>,
         "page split during eviction deepened the tree" : <num>,
         "modified pages evicted" : <num>,
         "pages selected for eviction unable to be evicted" : <num>,
         "pages evicted because they exceeded the in-memory maximum" : <num>,
         "pages evicted because they had chains of deleted items" : <num>,
         "failed eviction of pages that exceeded the in-memory maximum" : <num>,
         "hazard pointer blocked page eviction" : <num>,
         "internal pages evicted" : <num>,
         "maximum page size at eviction" : <num>,
         "eviction server candidate queue empty when topping up" : <num>,
         "eviction server candidate queue not empty when topping up" : <num>,
         "eviction server evicting pages" : <num>,
         "eviction server populating queue, but not evicting pages" : <num>,
         "eviction server unable to reach eviction goal" : <num>,
         "pages split during eviction" : <num>,
         "pages walked for eviction" : <num>,
         "eviction worker thread evicting pages" : <num>,
         "in-memory page splits" : <num>,
         "percentage overhead" : <num>,
         "tracked dirty pages in the cache" : <num>,
         "pages currently held in the cache" : <num>,
         "pages read into cache" : <num>,
         "pages written from cache" : <num>
   },
   "connection" : {
         "pthread mutex condition wait calls" : <num>,
         "files currently open" : <num>,
         "memory allocations" : <num>,
         "memory frees" : <num>,
         "memory re-allocations" : <num>,
         "total read I/Os" : <num>,
         "pthread mutex shared lock read-lock calls" : <num>,
         "pthread mutex shared lock write-lock calls" : <num>,
         "total write I/Os" : <num>
   },
   "cursor" : {
         "cursor create calls" : <num>,
         "cursor insert calls" : <num>,
         "cursor next calls" : <num>,
         "cursor prev calls" : <num>,
         "cursor remove calls" : <num>,
         "cursor reset calls" : <num>,
         "cursor search calls" : <num>,
         "cursor search near calls" : <num>,
         "cursor update calls" : <num>
   },
   "data-handle" : {
         "connection dhandles swept" : <num>,
         "connection candidate referenced" : <num>,
         "connection sweeps" : <num>,
         "connection time-of-death sets" : <num>,
         "session dhandles swept" : <num>,
         "session sweep attempts" : <num>
   },
   "log" : {
         "log buffer size increases" : <num>,
         "total log buffer size" : <num>,
         "log bytes of payload data" : <num>,
         "log bytes written" : <num>,
         "yields waiting for previous log file close" : <num>,
         "total size of compressed records" : <num>,
         "total in-memory size of compressed records" : <num>,
         "log records too small to compress" : <num>,
         "log records not compressed" : <num>,
         "log records compressed" : <num>,
         "maximum log file size" : <num>,
         "pre-allocated log files prepared" : <num>,
         "number of pre-allocated log files to create" : <num>,
         "pre-allocated log files used" : <num>,
         "log read operations" : <num>,
         "log release advances write LSN" : <num>,
         "records processed by log scan" : <num>,
         "log scan records requiring two reads" : <num>,
         "log scan operations" : <num>,
         "consolidated slot closures" : <num>,
         "logging bytes consolidated" : <num>,
         "consolidated slot joins" : <num>,
         "consolidated slot join races" : <num>,
         "slots selected for switching that were unavailable" : <num>,
         "record size exceeded maximum" : <num>,
         "failed to find a slot large enough for record" : <num>,
         "consolidated slot join transitions" : <num>,
         "log sync operations" : <num>,
         "log sync_dir operations" : <num>,
         "log server thread advances write LSN" : <num>,
         "log write operations" : <num>
   },
   "reconciliation" : {
         "page reconciliation calls" : <num>,
         "page reconciliation calls for eviction" : <num>,
         "split bytes currently awaiting free" : <num>,
         "split objects currently awaiting free" : <num>
   },
   "session" : {
         "open cursor count" : <num>,
         "open session count" : <num>
   },
   "thread-yield" : {
         "page acquire busy blocked" : <num>,
         "page acquire eviction blocked" : <num>,
         "page acquire locked blocked" : <num>,
         "page acquire read blocked" : <num>,
         "page acquire time sleeping (usecs)" : <num>
   },
   "transaction" : {
         "transaction begins" : <num>,
         "transaction checkpoints" : <num>,
         "transaction checkpoint currently running" : <num>,
         "transaction checkpoint max time (msecs)" : <num>,
         "transaction checkpoint min time (msecs)" : <num>,
         "transaction checkpoint most recent time (msecs)" : <num>,
         "transaction checkpoint total time (msecs)" : <num>,
         "transactions committed" : <num>,
         "transaction failures due to cache overflow" : <num>,
         "transaction range of IDs currently pinned" : <num>,
         "transactions rolled back" : <num>
   },
   "concurrentTransactions" : {
         "write" : {
            "out" : <num>,
            "available" : <num>,
            "totalTickets" : <num>
         },
         "read" : {
            "out" : <num>,
            "available" : <num>,
            "totalTickets" : <num>
         }
   }
},
```

The final `ok` field holds the return status for the `serverStatus` command:

```
"ok" : 1
```

3.5 Journaling

On this page

- Journaling and WiredTiger (page 147)
- Journaling and MMAPv1 (page 148)

To provide durability in the event of a failure, MongoDB uses *write ahead logging* to on-disk *journal* files.

3.5.1 Journaling and WiredTiger

Important: The `log` mentioned in this section refers to the WiredTiger write-ahead log and not the MongoDB log file.

WiredTiger uses *checkpoints* to provide a consistent view of data on disk and allow MongoDB to recover from the last checkpoint. However, if MongoDB exits unexpectedly in between checkpoints, journaling is required to recover information that occured after the last checkpoint.

With journaling, the recovery process:

1. Looks in the data files to find the identifier of the last checkpoint.
2. Searches in the journal files for the log record that match that identifier of the last checkpoint.
3. Apply the operations in the journal files since the last checkpoint.

Journaling Process

With journaling, WiredTiger creates a log record per transaction.

MongoDB configures WiredTiger to use in-memory buffering for storing the write-ahead log records. Threads coordinate to allocate and copy into their portion of the buffer. When the buffer becomes full, the buffer will be written to the backing file in the file system. If the system becomes idle such that the buffer does not fill, WiredTiger will periodically (several times a second) write any buffers that are in memory.

WiredTiger syncs journal files to disk according to the following intervals or conditions:

- MongoDB sets checkpoints to occur in WiredTiger on user data at an interval of 60 seconds or when 2 GB of journal data has been written, whichever occurs first.

- Because MongoDB uses a log file size limit of 100 MB, WiredTiger creates a new journal file approximately every 100MB of data. When WiredTiger creates a new journal file, WiredTiger syncs the previous journal file.
- If the write operation includes a write concern of `j:true`, WiredTiger forces a sync of the WiredTiger log files.

Important: In between write operations, while the log records remain in the WiredTiger buffers, updates can be lost following a hard shutdown of `mongod`.

See also:

The `serverStatus` command returns information on the WiredTiger log statistics in the `wiredTiger.log` field.

Journal Files

For the journal files, MongoDB creates a subdirectory named `journal` under the `dbPath` directory. WiredTiger journal files have names with the following format `WiredTigerLog.<sequence>` where `<sequence>` is a zero-padded number starting from `0000000001`.

Journal files contain the log records. Each log record has a unique identifier.

MongoDB configures WiredTiger to use snappy compression for the journaling data.

Minimum log record size for WiredTiger is 128 bytes. If a log record is 128 bytes or smaller, WiredTiger does not compress that record.

WiredTiger journal files for MongoDB have a maximum size limit of approximately 100 MB. Once the file exceeds that limit, WiredTiger creates a new journal file.

WiredTiger automatically removes old log files to maintain only the files needed to recover from last checkpoint.

WireTiger will pre-allocate log files.

3.5.2 Journaling and MMAPv1

With MMAPv1, when a write operation occurs, MongoDB updates the in-memory view. With journaling enabled, MongoDB writes the in-memory changes first to on-disk journal files. If MongoDB should terminate or encounter an error before committing the changes to the data files, MongoDB can use the journal files to apply the write operation to the data files and maintain a consistent state.

Journaling Process

With journaling, MongoDB's storage layer has two internal views of the data set: the *private view*, used to write to the journal files, and the *shared view*, used to write to the data files:

1. MongoDB first applies write operations to the private view.
2. MongoDB then applies the changes in the private view to the *journal files* (page 149) in the `journal` directory. MongoDB records the write operations to the journal in batches called group commits. Grouping the commits help minimize the performance impact of journaling since these commits must block all writers during the commit. Writes to the journal are atomic, ensuring the consistency of the on-disk journal files. For information on the frequency of the commit interval, see `storage.journal.commitIntervalMs`
3. Upon a journal commit, MongoDB applies the changes from the journal to the shared view.

4. Finally, MongoDB applies the changes in the shared view to the data files. More precisely, at default intervals of 60 seconds, MongoDB asks the operating system to flush the shared view to the data files. The operating system may choose to flush the shared view to disk at a higher frequency than 60 seconds, particularly if the system is low on free memory.

If the `mongod` instance were to crash without having applied the writes to the data files, the journal could replay the writes to the shared view for eventual write to the data files.

When MongoDB flushes write operations to the data files, MongoDB notes which journal writes have been flushed. Once a journal file contains only flushed writes, it is no longer needed for recovery and MongoDB can recycle it for a new journal file.

Once the journal operations have been applied to the shared view and flushed to disk (i.e. pages in the shared view and private view are in sync), MongoDB asks the operating system to remap the shared view to the private view in order to save physical RAM. MongoDB routinely asks the operating system to remap the shared view to the `private view` in order to save physical RAM. Upon a new remapping, the operating system knows that physical memory pages can be shared between the shared view and the private view mappings.

Note: The interaction between the shared view and the on-disk data files is similar to how MongoDB works *without* journaling. Without journaling, MongoDB asks the operating system to flush in-memory changes to the data files every 60 seconds.

Journal Files

With journaling enabled, MongoDB creates a subdirectory named `journal` under the `dbPath` directory. The `journal` directory contains journal files named `j._<sequence>` where `<sequence>` is an integer starting from `0` and a "last sequence number" file `lsn`.

Journal files contain the write ahead logs; each journal entry describes the bytes the write operation changed in the data files. Journal files are append-only files. When a journal file holds 1 gigabyte of data, MongoDB creates a new journal file. If you use the `storage.smallFiles` option when starting `mongod`, you limit the size of each journal file to 128 megabytes.

The `lsn` file contains the last time MongoDB flushed the changes to the data files.

Once MongoDB applies all the write operations in a particular journal file to the data files, MongoDB can recycle it for a new journal file.

Unless you write *many* bytes of data per second, the `journal` directory should contain only two or three journal files.

A clean shutdown removes all the files in the journal directory. A dirty shutdown (crash) leaves files in the journal directory; these are used to automatically recover the database to a consistent state when the mongod process is restarted.

Journal Directory

To speed the frequent sequential writes that occur to the current journal file, you can ensure that the journal directory is on a different filesystem from the database data files.

Important: If you place the journal on a different filesystem from your data files, you *cannot* use a filesystem snapshot alone to capture valid backups of a `dbPath` directory. In this case, use `fsyncLock()` to ensure that database files are consistent before the snapshot and `fsyncUnlock()` once the snapshot is complete.

Preallocation Lag

MongoDB may preallocate journal files if the `mongod` process determines that it is more efficient to preallocate journal files than create new journal files as needed.

Depending on your filesystem, you might experience a preallocation lag the first time you start a `mongod` instance with journaling enabled. The amount of time required to pre-allocate files might last several minutes; during this time, you will not be able to connect to the database. This is a one-time preallocation and does not occur with future invocations.

To avoid preallocation lag, see *Avoid Preallocation Lag for MMAPv1* (page 64).

3.6 Exit Codes and Statuses

MongoDB will return one of the following codes and statuses when exiting. Use this guide to interpret logs and when troubleshooting issues with `mongod` and `mongos` instances.

0

Returned by MongoDB applications upon successful exit.

2

The specified options are in error or are incompatible with other options.

3

Returned by `mongod` if there is a mismatch between hostnames specified on the command line and in the `local.sources` collection. `mongod` may also return this status if *oplog* collection in the `local` database is not readable.

4

The version of the database is different from the version supported by the `mongod` (or `mongod.exe`) instance. The instance exits cleanly. Restart `mongod` with the `--upgrade` option to upgrade the database to the version supported by this `mongod` instance.

5

Returned by `mongod` if a `moveChunk` operation fails to confirm a commit.

12

Returned by the `mongod.exe` process on Windows when it receives a Control-C, Close, Break or Shutdown event.

14

Returned by MongoDB applications which encounter an unrecoverable error, an uncaught exception or uncaught signal. The system exits without performing a clean shut down.

20

Message: `ERROR: wsastartup failed <reason>`

Returned by MongoDB applications on Windows following an error in the WSAStartup function.

Message: `NT Service Error`

Returned by MongoDB applications for Windows due to failures installing, starting or removing the NT Service for the application.

45

Returned when a MongoDB application cannot open a file or cannot obtain a lock on a file.

47

MongoDB applications exit cleanly following a large clock skew (32768 milliseconds) event.

48

`mongod` exits cleanly if the server socket closes. The server socket is on port `27017` by default, or as specified to the `--port` run-time option.

49

Returned by `mongod.exe` or `mongos.exe` on Windows when either receives a shutdown message from the *Windows Service Control Manager*.

100

Returned by `mongod` when the process throws an uncaught exception.

CHAPTER 4

Production Checklist

On this page

The following checklists provide recommendations that will help you avoid issues in your production MongoDB deployment.

4.1 Operations Checklist

On this page

The following checklist, along with the *Development* (page 156) list, provides recommendations to help you avoid issues in your production MongoDB deployment.

4.1.1 Filesystem

- Align your disk partitions with your RAID configuration.
- Avoid using NFS drives for your `dbPath`. Using NFS drives can result in degraded and unstable performance. See: *Remote Filesystems* (page 24) for more information.
  - VMWare users should use VMWare virtual drives over NFS.

- Linux/Unix: format your drives into XFS or EXT4. If possible, use XFS as it generally performs better with MongoDB.
  - With the WiredTiger storage engine, use of XFS is **strongly recommended** to avoid performance issues found when using EXT4 with WiredTiger.
  - If using RAID, you may need to configure XFS with your RAID geometry.
- Windows: use the NTFS file system. **Do not** use any FAT file system (i.e. FAT 16/32/exFAT).

4.1.2 Replication

- Verify that all non-hidden replica set members are identically provisioned in terms of their RAM, CPU, disk, network setup, etc.
- *Configure the oplog size* (page 187) to suit your use case:
  - The replication oplog window should cover normal maintenance and downtime windows to avoid the need for a full resync.
  - The replication oplog window should cover the time needed to restore a replica set member, either by an initial sync or by restoring from the last backup.
- Ensure that your replica set includes at least three data-bearing nodes with `w:majority write concern`. Three data-bearing nodes are required for replica set-wide data durability.
- Use hostnames when configuring replica set members, rather than IP addresses.
- Ensure full bidirectional network connectivity between all `mongod` instances.
- Ensure that each host can resolve itself.
- Ensure that your replica set contains an odd number of voting members.
- Ensure that `mongod` instances have `0` or `1` votes.
- For high availability, deploy your replica set into a *minimum* of three data centers.

4.1.3 Sharding

- Place your `config servers` on dedicated hardware for optimal performance in large clusters. Ensure that the hardware has enough RAM to hold the data files entirely in memory and that it has dedicated storage.
- Use NTP to synchronize the clocks on all components of your sharded cluster.
- Ensure full bidirectional network connectivity between `mongod`, `mongos` and config servers.
- Use CNAMEs to identify your config servers to the cluster so that you can rename and renumber your config servers without downtime.

4.1.4 Journaling: MMAPv1 Storage Engine

- Ensure that all instances use *journaling* (page 147).
- Place the journal on its own low-latency disk for write-intensive workloads. Note that this will affect snapshot-style backups as the files constituting the state of the database will reside on separate volumes.

4.1.5 Hardware

- Use RAID10 and SSD drives for optimal performance.
- SAN and Virtualization:
  - Ensure that each `mongod` has provisioned IOPS for its `dbPath`, or has its own physical drive or LUN.
  - Avoid dynamic memory features, such as memory ballooning, when running in virtual environments.
  - Avoid placing all replica set members on the same SAN, as the SAN can be a single point of failure.

4.1.6 Deployments to Cloud Hardware

- Windows Azure: Adjust the TCP keepalive (`tcp_keepalive_time`) to 100-120. The default TTL for TCP connections on Windows Azure load balancers is too slow for MongoDB's connection pooling behavior.
- Use MongoDB version 2.6.4 or later on systems with high-latency storage, such as Windows Azure, as these versions include performance improvements for those systems. See: Azure Deployment Recommendations[1] for more information.

4.1.7 Operating System Configuration

Linux

- Turn off transparent hugepages and defrag. See *Transparent Huge Pages Settings* (page 48) for more information.
- *Adjust the readahead settings* (page 26) on the devices storing your database files to suit your use case. If your working set is bigger that the available RAM, and the document access pattern is random, consider lowering the readahead to 32 or 16. Evaluate different settings to find an optimal value that maximizes the resident memory and lowers the number of page faults.
- Use the `noop` or `deadline` disk schedulers for SSD drives.
- Use the `noop` disk scheduler for virtualized drives in guest VMs.
- Disable NUMA or set vm.zone_reclaim_mode to 0 and run `mongod` instances with node interleaving. See: *MongoDB and NUMA Hardware* (page 22) for more information.
- Adjust the `ulimit` values on your hardware to suit your use case. If multiple `mongod` or `mongos` instances are running under the same user, scale the `ulimit` values accordingly. See: *UNIX ulimit Settings* (page 125) for more information.
- Use `noatime` for the `dbPath` mount point.
- Configure sufficient file handles (`fs.file-max`), kernel pid limit (`kernel.pid_max`), and maximum threads per process (`kernel.threads-max`) for your deployment. For large systems, values of 98000, 32768, and 64000 are a good starting point.
- Ensure that your system has swap space configured. Refer to your operating system's documentation for details on appropriate sizing.
- Ensure that the system default TCP keepalive is set correctly. A value of 300 often provides better performance for replica sets and sharded clusters. See: *faq-keepalive* in the Frequently Asked Questions for more information.

[1] https://docs.mongodb.org/ecosystem/platforms/windows-azure

Windows

- Consider disabling NTFS “last access time” updates. This is analogous to disabling `atime` on Unix-like systems.

4.1.8 Backups

- Schedule periodic tests of your back up and restore process to have time estimates on hand, and to verify its functionality.

4.1.9 Monitoring

- Use MongoDB Cloud Manager[2] or Ops Manager, an on-premise solution available in MongoDB Enterprise Advanced[3] or another monitoring system to monitor key database metrics and set up alerts for them. Include alerts for the following metrics:
  - lock percent (for the *MMAPv1 storage engine*)
  - replication lag
  - replication oplog window
  - assertions
  - queues
  - page faults
- Monitor hardware statistics for your servers. In particular, pay attention to the disk use, CPU, and available disk space.

  In the absence of disk space monitoring, or as a precaution:
  - Create a dummy 4GB file on the `storage.dbPath` drive to ensure available space if the disk becomes full.
  - A combination of `cron+df` can alert when disk space hits a high-water mark, if no other monitoring tool is available.

4.1.10 Load Balancing

- Configure load balancers to enable “sticky sessions” or “client affinity”, with a sufficient timeout for existing connections.
- Avoid placing load balancers between MongoDB cluster or replica set components.

4.2 Development

[2] https://cloud.mongodb.com/?jmp=docs
[3] https://www.mongodb.com/products/mongodb-enterprise-advanced?jmp=docs

On this page

The following checklist, along with the *Operations Checklist* (page 153), provides recommendations to help you avoid issues in your production MongoDB deployment.

4.2.1 Data Durability

- Ensure that your replica set includes at least three data-bearing nodes with `w:majority write concern`. Three data-bearing nodes are required for replica-set wide data durability.
- Ensure that all instances use *journaling* (page 147).

4.2.2 Schema Design

- Ensure that your schema design does not rely on indexed arrays that grow in length without bound. Typically, best performance can be achieved when such indexed arrays have fewer than 1000 elements.

4.2.3 Replication

- Do not use secondary reads to scale overall read throughput. See: Can I use more replica nodes to scale[4] for an overview of read scaling. For information about secondary reads, see: `https://docs.mongodb.org/manual/core/read-preference`.

4.2.4 Sharding

- Ensure that your shard key distributes the load evenly on your shards. See: *Considerations for Selecting Shard Keys* (page 216) for more information.
- Use `targeted queries` for workloads that need to scale with the number of shards.
- Always read from primary nodes for non-targeted queries that may be sensitive to stale or orphaned data[5].
- *Pre-split and manually balance chunks* (page 246) when inserting large data sets into a new non-hashed sharded collection. Pre-splitting and manually balancing enables the insert load to be distributed among the shards, increasing performance for the initial load.

4.2.5 Drivers

- Make use of connection pooling. Most MongoDB drivers support connection pooling. Adjust the connection pool size to suit your use case, beginning at 110-115% of the typical number of concurrent database requests.

[4] http://askasya.com/post/canreplicashelpscaling

[5] http://blog.mongodb.org/post/74730554385/background-indexing-on-secondaries-and-orphaned

- Ensure that your applications handle transient write and read errors during replica set elections.
- Ensure that your applications handle failed requests and retry them if applicable. Drivers **do not** automatically retry failed requests.
- Use exponential backoff logic for database request retries.
- Use `cursor.maxTimeMS()` for reads and *wc-wtimeout* for writes if you need to cap execution time for database operations.

4.3 Additional Resources

- MongoDB Production Readiness Consulting Package[6]
- MongoDB Ops Optimization Consulting Package[7]

[6] https://www.mongodb.com/products/consulting?jmp=docs#s_product_readiness
[7] https://www.mongodb.com/products/consulting?jmp=docs#ops_optimization

CHAPTER 5

Appendix

5.1 Replica Set Tutorials

The administration of *replica sets* includes the initial deployment of the set, adding and removing members to a set, and configuring the operational parameters and properties of the set. Administrators generally need not intervene in failover or replication processes as MongoDB automates these functions. In the exceptional situations that require manual interventions, the tutorials in these sections describe processes such as resyncing a member. The tutorials in this section form the basis for all replica set administration.

***Replica Set Deployment Tutorials* (page 160)** Instructions for deploying replica sets, as well as adding and removing members from an existing replica set.

***Deploy a Replica Set* (page 160)** Configure a three-member replica set for production systems.

***Convert a Standalone to a Replica Set* (page 172)** Convert an existing standalone `mongod` instance into a three-member replica set.

***Add Members to a Replica Set* (page 173)** Add a new member to an existing replica set.

***Remove Members from Replica Set* (page 176)** Remove a member from a replica set.

Continue reading from *Replica Set Deployment Tutorials* (page 160) for additional tutorials of related to setting up replica set deployments.

***Member Configuration Tutorials* (page 178)** Tutorials that describe the process for configuring replica set members.

***Adjust Priority for Replica Set Member* (page 179)** Change the precedence given to a replica set members in an election for primary.

***Prevent Secondary from Becoming Primary* (page 180)** Make a secondary member ineligible for election as primary.

***Configure a Hidden Replica Set Member* (page 181)** Configure a secondary member to be invisible to applications in order to support significantly different usage, such as a dedicated backups.

Continue reading from *Member Configuration Tutorials* (page 178) for more tutorials that describe replica set configuration.

***Replica Set Maintenance Tutorials* (page 187)** Procedures and tasks for common operations on active replica set deployments.

***Change the Size of the Oplog* (page 187)** Increase the size of the *oplog* which logs operations. In most cases, the default oplog size is sufficient.

***Resync a Member of a Replica Set* (page 193)** Sync the data on a member. Either perform initial sync on a new member or resync the data on an existing member that has fallen too far behind to catch up by way of normal replication.

***Force a Member to Become Primary* (page 191)** Force a replica set member to become primary.

***Change Hostnames in a Replica Set* (page 202)** Update the replica set configuration to reflect changes in members' hostnames.

Continue reading from *Replica Set Maintenance Tutorials* (page 187) for descriptions of additional replica set maintenance procedures.

***Troubleshoot Replica Sets* (page 207)** Describes common issues and operational challenges for replica sets. For additional diagnostic information, see `https://docs.mongodb.org/manual/faq/diagnostics`.

5.1.1 Replica Set Deployment Tutorials

The following tutorials provide information in deploying replica sets.

***Deploy a Replica Set* (page 160)** Configure a three-member replica set for production systems.

***Deploy a Replica Set for Testing and Development* (page 163)** Configure a three-member replica set for either development or testing systems.

***Deploy a Geographically Redundant Replica Set* (page 165)** Create a geographically redundant replica set to protect against location-centered availability limitations (e.g. network and power interruptions).

***Add an Arbiter to Replica Set* (page 171)** Add an arbiter give a replica set an odd number of voting members to prevent election ties.

***Convert a Standalone to a Replica Set* (page 172)** Convert an existing standalone `mongod` instance into a three-member replica set.

***Add Members to a Replica Set* (page 173)** Add a new member to an existing replica set.

***Remove Members from Replica Set* (page 176)** Remove a member from a replica set.

***Replace a Replica Set Member* (page 178)** Update the replica set configuration when the hostname of a member's corresponding `mongod` instance has changed.

Deploy a Replica Set

On this page

This tutorial describes how to create a three-member *replica set* from three existing `mongod` instances running with `access control` disabled.

To deploy a replica set with enabled `access control`, see *deploy-repl-set-with-auth*. If you wish to deploy a replica set from a single MongoDB instance, see *Convert a Standalone to a Replica Set* (page 172). For more information on replica set deployments, see the `https://docs.mongodb.org/manual/replication` and `https://docs.mongodb.org/manual/core/replica-set-architectures` documentation.

Overview

Three member *replica sets* provide enough redundancy to survive most network partitions and other system failures. These sets also have sufficient capacity for many distributed read operations. Replica sets should always have an odd number of members. This ensures that `elections` will proceed smoothly. For more about designing replica sets, see `the Replication overview`.

The basic procedure is to start the `mongod` instances that will become members of the replica set, configure the replica set itself, and then add the `mongod` instances to it.

Requirements

For production deployments, you should maintain as much separation between members as possible by hosting the `mongod` instances on separate machines. When using virtual machines for production deployments, you should place each `mongod` instance on a separate host server serviced by redundant power circuits and redundant network paths.

Before you can deploy a replica set, you must install MongoDB on each system that will be part of your *replica set*. If you have not already installed MongoDB, see the *installation tutorials*.

Before creating your replica set, you should verify that your network configuration allows all possible connections between each member. For a successful replica set deployment, every member must be able to connect to every other member. For instructions on how to check your connection, see *Test Connections Between all Members* (page 209).

Considerations When Deploying a Replica Set

Architecture In a production, deploy each member of the replica set to its own machine and if possible bind to the standard MongoDB port of `27017`. Use the `bind_ip` option to ensure that MongoDB listens for connections from applications on configured addresses.

For a geographically distributed replica sets, ensure that the majority of the set's `mongod` instances reside in the primary site.

See `https://docs.mongodb.org/manual/core/replica-set-architectures` for more information.

Connectivity Ensure that network traffic can pass between all members of the set and all clients in the network securely and efficiently. Consider the following:

- Establish a virtual private network. Ensure that your network topology routes all traffic between members within a single site over the local area network.
- Configure access control to prevent connections from unknown clients to the replica set.
- Configure networking and firewall rules so that incoming and outgoing packets are permitted only on the default MongoDB port and only from within your deployment.

Finally ensure that each member of a replica set is accessible by way of resolvable DNS or hostnames. You should either configure your DNS names appropriately or set up your systems' `/etc/hosts` file to reflect this configuration.

Configuration Specify the run time configuration on each system in a `configuration file` stored in `/etc/mongod.conf` or a related location. Create the directory where MongoDB stores data files before deploying MongoDB.

For more information about the run time options used above and other configuration options, see `https://docs.mongodb.org/manual/reference/configuration-options`.

Procedure

The following procedure outlines the steps to deploy a replica set when access control is disabled.

Step 1: Start each member of the replica set with the appropriate options. For each member, start a `mongod` and specify the replica set name through the `replSet` option. Specify any other parameters specific to your deployment. For replication-specific parameters, see *cli-mongod-replica-set*.

If your application connects to more than one replica set, each set should have a distinct name. Some drivers group replica set connections by replica set name.

The following example specifies the replica set name through the *`--replSet`* command-line option:

```
mongod --replSet "rs0"
```

You can also specify the `replica set name` in the `configuration file`. To start `mongod` with a configuration file, specify the file with the *`--config`* option:

```
mongod --config $HOME/.mongodb/config
```

In production deployments, you can configure a *init script* to manage this process. Init scripts are beyond the scope of this document.

Step 2: Connect a `mongo` shell to a replica set member. For example, to connect to a `mongod` running on localhost on the default port of `27017`, simply issue:

```
mongo
```

Step 3: Initiate the replica set. Use `rs.initiate()` on the replica set member:

```
rs.initiate()
```

MongoDB initiates a set that consists of the current member and that uses the default replica set configuration.

Step 4: Verify the initial replica set configuration. Use `rs.conf()` to display the `replica set configuration object`:

```
rs.conf()
```

The replica set configuration object resembles the following:

```
{
   "_id" : "rs0",
   "version" : 1,
   "members" : [
      {
         "_id" : 1,
         "host" : "mongodb0.example.net:27017"
      }
   ]
}
```

Step 5: Add the remaining members to the replica set. Add the remaining members with the `rs.add()` method.

The following example adds two members:

```
rs.add("mongodb1.example.net")
rs.add("mongodb2.example.net")
```

When complete, you have a fully functional replica set. The new replica set will elect a *primary*.

Step 6: Check the status of the replica set. Use the `rs.status()` operation:

```
rs.status()
```

See also:

deploy-repl-set-with-auth

Deploy a Replica Set for Testing and Development

On this page

This procedure describes deploying a replica set in a development or test environment. For a production deployment, refer to the *Deploy a Replica Set* (page 160) tutorial.

This tutorial describes how to create a three-member *replica set* from three existing `mongod` instances running with `access control` disabled.

To deploy a replica set with enabled `access control`, see *deploy-repl-set-with-auth*. If you wish to deploy a replica set from a single MongoDB instance, see *Convert a Standalone to a Replica Set* (page 172). For more information on replica set deployments, see the `https://docs.mongodb.org/manual/replication` and `https://docs.mongodb.org/manual/core/replica-set-architectures` documentation.

Overview

Three member *replica sets* provide enough redundancy to survive most network partitions and other system failures. These sets also have sufficient capacity for many distributed read operations. Replica sets should always have an odd number of members. This ensures that `elections` will proceed smoothly. For more about designing replica sets, see `the Replication overview`.

The basic procedure is to start the `mongod` instances that will become members of the replica set, configure the replica set itself, and then add the `mongod` instances to it.

Requirements

For test and development systems, you can run your `mongod` instances on a local system, or within a virtual instance.

Before you can deploy a replica set, you must install MongoDB on each system that will be part of your *replica set*. If you have not already installed MongoDB, see the *installation tutorials*.

Before creating your replica set, you should verify that your network configuration allows all possible connections between each member. For a successful replica set deployment, every member must be able to connect to every other member. For instructions on how to check your connection, see *Test Connections Between all Members* (page 209).

Considerations

Replica Set Naming

Important: These instructions should only be used for test or development deployments.

The examples in this procedure create a new replica set named `rs0`.

If your application connects to more than one replica set, each set should have a distinct name. Some drivers group replica set connections by replica set name.

You will begin by starting three `mongod` instances as members of a replica set named `rs0`.

Procedure

1. Create the necessary data directories for each member by issuing a command similar to the following:

   ```
   mkdir -p /srv/mongodb/rs0-0 /srv/mongodb/rs0-1 /srv/mongodb/rs0-2
   ```

   This will create directories called “rs0-0”, “rs0-1”, and “rs0-2”, which will contain the instances’ database files.

2. Start your `mongod` instances in their own shell windows by issuing the following commands:

   First member:

   ```
   mongod --port 27017 --dbpath /srv/mongodb/rs0-0 --replSet rs0 --smallfiles --oplogSize 128
   ```

   Second member:

   ```
   mongod --port 27018 --dbpath /srv/mongodb/rs0-1 --replSet rs0 --smallfiles --oplogSize 128
   ```

   Third member:

   ```
   mongod --port 27019 --dbpath /srv/mongodb/rs0-2 --replSet rs0 --smallfiles --oplogSize 128
   ```

   This starts each instance as a member of a replica set named `rs0`, each running on a distinct port, and specifies the path to your data directory with the *`--dbpath`* setting. If you are already using the suggested ports, select different ports.

   The *`--smallfiles`* and *`--oplogSize`* settings reduce the disk space that each `mongod` instance uses. This is ideal for testing and development deployments as it prevents overloading your machine. For more information on these and other configuration options, see `https://docs.mongodb.org/manual/reference/configuration-options`.

3. Connect to one of your `mongod` instances through the `mongo` shell. You will need to indicate which instance by specifying its port number. For the sake of simplicity and clarity, you may want to choose the first one, as in the following command;

   ```
   mongo --port 27017
   ```

4. In the `mongo` shell, use `rs.initiate()` to initiate the replica set. You can create a replica set configuration object in the `mongo` shell environment, as in the following example:

   ```
   rsconf = {
               _id: "rs0",
               members: [
   ```

```
                        {
                         _id: 0,
                         host: "<hostname>:27017"
                        }
                      ]
          }
```

replacing `<hostname>` with your system's hostname, and then pass the `rsconf` file to `rs.initiate()` as follows:

```
rs.initiate( rsconf )
```

5. Display the current `replica configuration` by issuing the following command:

   ```
   rs.conf()
   ```

   The replica set configuration object resembles the following

   ```
   {
      "_id" : "rs0",
      "version" : 4,
      "members" : [
         {
            "_id" : 1,
            "host" : "localhost:27017"
         }
      ]
   }
   ```

6. In the `mongo` shell connected to the *primary*, add the second and third `mongod` instances to the replica set using the `rs.add()` method. Replace `<hostname>` with your system's hostname in the following examples:

   ```
   rs.add("<hostname>:27018")
   rs.add("<hostname>:27019")
   ```

   When complete, you should have a fully functional replica set. The new replica set will elect a *primary*.

Check the status of your replica set at any time with the `rs.status()` operation.

See also:

The documentation of the following shell functions for more information:

- `rs.initiate()`
- `rs.conf()`
- `rs.reconfig()`
- `rs.add()`

You may also consider the simple setup script[1] as an example of a basic automatically-configured replica set.

Refer to `Replica Set Read and Write Semantics` for a detailed explanation of read and write semantics in MongoDB.

Deploy a Geographically Redundant Replica Set

[1] https://github.com/mongodb/mongo-snippets/blob/master/replication/simple-setup.py

On this page

Overview

This tutorial outlines the process for deploying a *replica set* with members in multiple locations. The tutorial addresses three-member sets, four-member sets, and sets with more than four members.

For appropriate background, see `https://docs.mongodb.org/manual/replication` and `https://docs.mongodb.org/manual/core/replica-set-architectures`. For related tutorials, see *Deploy a Replica Set* (page 160) and *Add Members to a Replica Set* (page 173).

Considerations

While *replica sets* provide basic protection against single-instance failure, replica sets whose members are all located in a single facility are susceptible to errors in that facility. Power outages, network interruptions, and natural disasters are all issues that can affect replica sets whose members are colocated. To protect against these classes of failures, deploy a replica set with one or more members in a geographically distinct facility or data center to provide redundancy.

Prerequisites

In general, the requirements for any geographically redundant replica set are as follows:

- Ensure that a majority of the *voting members* are within a primary facility, "Site A". This includes `priority 0 members` and `arbiters`. Deploy other members in secondary facilities, "Site B", "Site C", etc., to provide additional copies of the data. See *determine-geographic-distribution* for more information on the voting requirements for geographically redundant replica sets.
- If you deploy a replica set with an even number of members, deploy an `arbiter` on Site A. The arbiter must be on site A to keep the majority there.

For instance, for a three-member replica set you need two instances in a Site A, and one member in a secondary facility, Site B. Site A should be the same facility or very close to your primary application infrastructure (i.e. application servers, caching layer, users, etc.)

A four-member replica set should have at least two members in Site A, with the remaining members in one or more secondary sites, as well as a single *arbiter* in Site A.

For all configurations in this tutorial, deploy each replica set member on a separate system. Although you may deploy more than one replica set member on a single system, doing so reduces the redundancy and capacity of the replica set. Such deployments are typically for testing purposes and beyond the scope of this tutorial.

This tutorial assumes you have installed MongoDB on each system that will be part of your replica set. If you have not already installed MongoDB, see the *installation tutorials*.

Procedures

General Considerations

Architecture In a production, deploy each member of the replica set to its own machine and if possible bind to the standard MongoDB port of `27017`. Use the `bind_ip` option to ensure that MongoDB listens for connections from applications on configured addresses.

For a geographically distributed replica sets, ensure that the majority of the set's `mongod` instances reside in the primary site.

See `https://docs.mongodb.org/manual/core/replica-set-architectures` for more information.

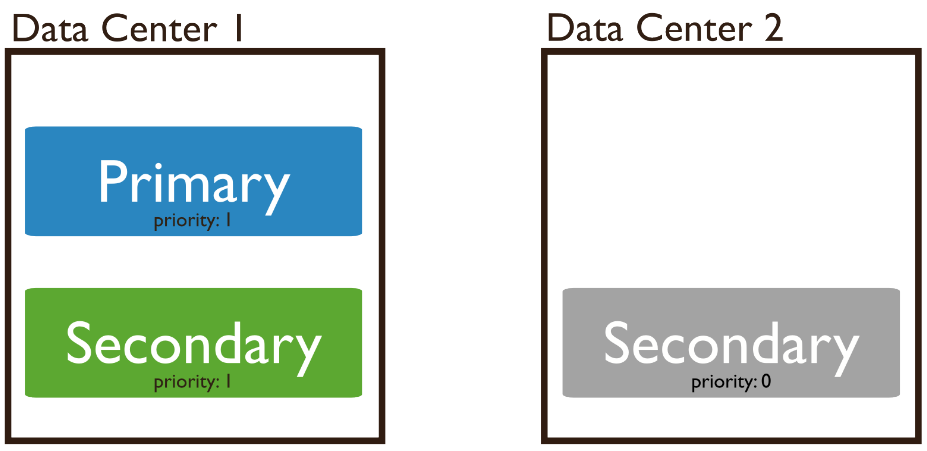
Connectivity Ensure that network traffic can pass between all members of the set and all clients in the network securely and efficiently. Consider the following:

- Establish a virtual private network. Ensure that your network topology routes all traffic between members within a single site over the local area network.
- Configure access control to prevent connections from unknown clients to the replica set.
- Configure networking and firewall rules so that incoming and outgoing packets are permitted only on the default MongoDB port and only from within your deployment.

Finally ensure that each member of a replica set is accessible by way of resolvable DNS or hostnames. You should either configure your DNS names appropriately or set up your systems' `/etc/hosts` file to reflect this configuration.

Configuration Specify the run time configuration on each system in a `configuration file` stored in `/etc/mongod.conf` or a related location. Create the directory where MongoDB stores data files before deploying MongoDB.

For more information about the run time options used above and other configuration options, see `https://docs.mongodb.org/manual/reference/configuration-options`.



Deploy a Geographically Redundant Three-Member Replica Set

Step 1: Start each member of the replica set with the appropriate options. For each member, start a `mongod` and specify the replica set name through the `replSet` option. Specify any other parameters specific to your deployment. For replication-specific parameters, see *cli-mongod-replica-set*.

If your application connects to more than one replica set, each set should have a distinct name. Some drivers group replica set connections by replica set name.

The following example specifies the replica set name through the *--replSet* command-line option:

```
mongod --replSet "rs0"
```

You can also specify the `replica set name` in the `configuration file`. To start `mongod` with a configuration file, specify the file with the *--config* option:

```
mongod --config $HOME/.mongodb/config
```

In production deployments, you can configure a *init script* to manage this process. Init scripts are beyond the scope of this document.

Step 2: Connect a `mongo` shell to a replica set member. For example, to connect to a `mongod` running on localhost on the default port of `27017`, simply issue:

```
mongo
```

Step 3: Initiate the replica set. Use `rs.initiate()` on the replica set member:

```
rs.initiate()
```

MongoDB initiates a set that consists of the current member and that uses the default replica set configuration.

Step 4: Verify the initial replica set configuration. Use `rs.conf()` to display the `replica set configuration object`:

```
rs.conf()
```

The replica set configuration object resembles the following:

```
{
   "_id" : "rs0",
   "version" : 1,
   "members" : [
      {
         "_id" : 1,
         "host" : "mongodb0.example.net:27017"
      }
   ]
}
```

Step 5: Add the remaining members to the replica set. Add the remaining members with the `rs.add()` method.

The following example adds two members:

```
rs.add("mongodb1.example.net")
rs.add("mongodb2.example.net")
```

When complete, you have a fully functional replica set. The new replica set will elect a *primary*.

Step 6: Configure the outside member as *priority 0 members*. Configure the member located in Site B (in this example, `mongodb2.example.net`) as a *priority 0 member*.

1. View the replica set configuration to determine the `members` array position for the member. Keep in mind the array position is not the same as the `_id`:

```
rs.conf()
```

2. Copy the replica set configuration object to a variable (to `cfg` in the example below). Then, in the variable, set the correct priority for the member. Then pass the variable to `rs.reconfig()` to update the replica set configuration.

   For example, to set priority for the third member in the array (i.e., the member at position 2), issue the following sequence of commands:

   ```
   cfg = rs.conf()
   cfg.members[2].priority = 0
   rs.reconfig(cfg)
   ```

   Note: The `rs.reconfig()` shell method can force the current primary to step down, causing an election. When the primary steps down, all clients will disconnect. This is the intended behavior. While most elections complete within a minute, always make sure any replica configuration changes occur during scheduled maintenance periods.

After these commands return, you have a geographically redundant three-member replica set.

Step 7: Check the status of the replica set. Use the `rs.status()` operation:

```
rs.status()
```

Deploy a Geographically Redundant Four-Member Replica Set A geographically redundant four-member deployment has two additional considerations:

- One host (e.g. `mongodb4.example.net`) must be an *arbiter*. This host can run on a system that is also used for an application server or on the same machine as another MongoDB process.
- You must decide how to distribute your systems. There are three possible architectures for the four-member replica set:
  - Three members in Site A, one *priority 0 member* in Site B, and an arbiter in Site A.
  - Two members in Site A, two *priority 0 members* in Site B, and an arbiter in Site A.
  - Two members in Site A, one priority 0 member in Site B, one priority 0 member in Site C, and an arbiter in site A.

  In most cases, the first architecture is preferable because it is the least complex.

To deploy a geographically redundant four-member set:

Step 1: Start each member of the replica set with the appropriate options. For each member, start a `mongod` and specify the replica set name through the `replSet` option. Specify any other parameters specific to your deployment. For replication-specific parameters, see *cli-mongod-replica-set*.

If your application connects to more than one replica set, each set should have a distinct name. Some drivers group replica set connections by replica set name.

The following example specifies the replica set name through the *`--replSet`* command-line option:

```
mongod --replSet "rs0"
```

You can also specify the `replica set name` in the `configuration file`. To start `mongod` with a configuration file, specify the file with the `--config` option:

```
mongod --config $HOME/.mongodb/config
```

In production deployments, you can configure a *init script* to manage this process. Init scripts are beyond the scope of this document.

Step 2: Connect a `mongo` shell to a replica set member. For example, to connect to a `mongod` running on localhost on the default port of `27017`, simply issue:

```
mongo
```

Step 3: Initiate the replica set. Use `rs.initiate()` on the replica set member:

```
rs.initiate()
```

MongoDB initiates a set that consists of the current member and that uses the default replica set configuration.

Step 4: Verify the initial replica set configuration. Use `rs.conf()` to display the `replica set configuration object`:

```
rs.conf()
```

The replica set configuration object resembles the following:

```
{
   "_id" : "rs0",
   "version" : 1,
   "members" : [
      {
         "_id" : 1,
         "host" : "mongodb0.example.net:27017"
      }
   ]
}
```

Step 5: Add the remaining members to the replica set. Use `rs.add()` in a `mongo` shell connected to the current primary. The commands should resemble the following:

```
rs.add("mongodb1.example.net")
rs.add("mongodb2.example.net")
rs.add("mongodb3.example.net")
```

When complete, you should have a fully functional replica set. The new replica set will elect a *primary*.

Step 6: Add the arbiter. In the same shell session, issue the following command to add the arbiter (e.g. `mongodb4.example.net`):

```
rs.addArb("mongodb4.example.net")
```

Step 7: Configure outside members as *priority 0 members*. Configure each member located outside of Site A (e.g. `mongodb3.example.net`) as a *priority 0 member*.

1. View the replica set configuration to determine the `members` array position for the member. Keep in mind the array position is not the same as the `_id`:

   ```
   rs.conf()
   ```

2. Copy the replica set configuration object to a variable (to `cfg` in the example below). Then, in the variable, set the correct priority for the member. Then pass the variable to `rs.reconfig()` to update the replica set configuration.

   For example, to set priority for the third member in the array (i.e., the member at position 2), issue the following sequence of commands:

   ```
   cfg = rs.conf()
   cfg.members[2].priority = 0
   rs.reconfig(cfg)
   ```

   Note: The `rs.reconfig()` shell method can force the current primary to step down, causing an election. When the primary steps down, all clients will disconnect. This is the intended behavior. While most elections complete within a minute, always make sure any replica configuration changes occur during scheduled maintenance periods.

After these commands return, you have a geographically redundant four-member replica set.

Step 8: Check the status of the replica set. Use the `rs.status()` operation:

```
rs.status()
```

Deploy a Geographically Redundant Set with More than Four Members The above procedures detail the steps necessary for deploying a geographically redundant replica set. Larger replica set deployments follow the same steps, but have additional considerations:

- Never deploy more than seven voting members.
- If you have an even number of members, use *the procedure for a four-member set* (page 169)). Ensure that a single facility, "Site A", always has a majority of the members by deploying the *arbiter* in that site. For example, if a set has six members, deploy at least three voting members in addition to the arbiter in Site A, and the remaining members in alternate sites.
- If you have an odd number of members, use *the procedure for a three-member set* (page 167). Ensure that a single facility, "Site A" always has a majority of the members of the set. For example, if a set has five members, deploy three members within Site A and two members in other facilities.
- If you have a majority of the members of the set *outside* of Site A and the network partitions to prevent communication between sites, the current primary in Site A will step down, even if none of the members outside of Site A are eligible to become primary.

Add an Arbiter to Replica Set

On this page

Arbiters are `mongod` instances that are part of a *replica set* but do not hold data. Arbiters participate in *elections* in order to break ties. If a replica set has an even number of members, add an arbiter.

Arbiters have minimal resource requirements and do not require dedicated hardware. You can deploy an arbiter on an application server or a monitoring host.

Important: Do not run an arbiter on the same system as a member of the replica set.

Considerations

An arbiter does not store data, but until the arbiter's `mongod` process is added to the replica set, the arbiter will act like any other `mongod` process and start up with a set of data files and with a full-sized *journal*.

To minimize the default creation of data, set the following in the arbiter's `configuration file`:

- `journal.enabled` to `false`

  Warning: Never set `journal.enabled` to `false` on a data-bearing node.

- `smallFiles` to `true`

These settings are specific to arbiters. Do not set `journal.enabled` to `false` on a data-bearing node. Similarly, do not set `smallFiles` unless specifically indicated.

Add an Arbiter

1. Create a data directory (e.g. `dbPath`) for the arbiter. The `mongod` instance uses the directory for configuration data. The directory *will not* hold the data set. For example, create the `/data/arb` directory:

   ```
   mkdir /data/arb
   ```

2. Start the arbiter. Specify the data directory and the replica set name. The following, starts an arbiter using the `/data/arb` `dbPath` for the `rs` replica set:

   ```
   mongod --port 30000 --dbpath /data/arb --replSet rs
   ```

3. Connect to the primary and add the arbiter to the replica set. Use the `rs.addArb()` method, as in the following example:

   ```
   rs.addArb("m1.example.net:30000")
   ```

   This operation adds the arbiter running on port `30000` on the `m1.example.net` host.

Convert a Standalone to a Replica Set

On this page

This tutorial describes the process for converting a *standalone* `mongod` instance into a three-member *replica set*. Use standalone instances for testing and development, but always use replica sets in production. To install a standalone instance, see the *installation tutorials*.

To deploy a replica set without using a pre-existing `mongod` instance, see *Deploy a Replica Set* (page 160).

Procedure

1. Shut down the *standalone* mongod instance.
2. Restart the instance. Use the *--replSet* option to specify the name of the new replica set.

   For example, the following command starts a standalone instance as a member of a new replica set named rs0. The command uses the standalone's existing database path of /srv/mongodb/db0:

   ```
   mongod --port 27017 --dbpath /srv/mongodb/db0 --replSet rs0
   ```

   If your application connects to more than one replica set, each set should have a distinct name. Some drivers group replica set connections by replica set name.

   For more information on configuration options, see https://docs.mongodb.org/manual/reference/configurat and the mongod manual page.
3. Connect to the mongod instance.
4. Use rs.initiate() to initiate the new replica set:

   ```
   rs.initiate()
   ```

   The replica set is now operational.

   To view the replica set configuration, use rs.conf(). To check the status of the replica set, use rs.status().

Expand the Replica Set Add additional replica set members by doing the following:

1. On two distinct systems, start two new standalone mongod instances. For information on starting a standalone instance, see the *installation tutorial* specific to your environment.
2. On your connection to the original mongod instance (the former standalone instance), issue a command in the following form for each new instance to add to the replica set:

   ```
   rs.add("<hostname><:port>")
   ```

   Replace <hostname> and <port> with the resolvable hostname and port of the mongod instance to add to the set. For more information on adding a host to a replica set, see *Add Members to a Replica Set* (page 173).

Sharding Considerations If the new replica set is part of a *sharded cluster*, change the shard host information in the *config database* by doing the following:

1. Connect to one of the sharded cluster's mongos instances and issue a command in the following form:

   ```
   db.getSiblingDB("config").shards.save( {_id: "<name>", host: "<replica-set>/<member,><member,><.
   ```

   Replace <name> with the name of the shard. Replace <replica-set> with the name of the replica set. Replace <member,><member,><> with the list of the members of the replica set.
2. Restart all mongos instances. If possible, restart all components of the replica sets (i.e., all mongos and all shard mongod instances).

Add Members to a Replica Set

On this page

Overview

This tutorial explains how to add an additional member to an existing *replica set*. For background on replication deployment patterns, see the `https://docs.mongodb.org/manual/core/replica-set-architectures` document.

Maximum Voting Members A replica set can have a maximum of seven *voting members*. To add a member to a replica set that already has seven voting members, you must either add the member as a *non-voting member* or remove a vote from an `existing member`.

Init Scripts In production deployments you can configure a *init script* to manage member processes.

Existing Members You can use these procedures to add new members to an existing set. You can also use the same procedure to "re-add" a removed member. If the removed member's data is still relatively recent, it can recover and catch up easily.

Data Files If you have a backup or snapshot of an existing member, you can move the data files (e.g. the `dbPath` directory) to a new system and use them to quickly initiate a new member. The files must be:

- A valid copy of the data files from a member of the same replica set. See *Backup and Restore with Filesystem Snapshots* (page 75) document for more information.

  Important: Always use filesystem snapshots to create a copy of a member of the existing replica set. **Do not** use `mongodump` and `mongorestore` to seed a new replica set member.

- More recent than the oldest operation in the *primary's oplog*. The new member must be able to become current by applying operations from the primary's oplog.

Requirements

1. An active replica set.
2. A new MongoDB system capable of supporting your data set, accessible by the active replica set through the network.

Otherwise, use the MongoDB *installation tutorial* and the *Deploy a Replica Set* (page 160) tutorials.

Procedures

Prepare the Data Directory Before adding a new member to an existing *replica set*, prepare the new member's *data directory* using one of the following strategies:

- Make sure the new member's data directory *does not* contain data. The new member will copy the data from an existing member.

  If the new member is in a *recovering* state, it must exit and become a *secondary* before MongoDB can copy all data as part of the replication process. This process takes time but does not require administrator intervention.

- Manually copy the data directory from an existing member. The new member becomes a secondary member and will catch up to the current state of the replica set. Copying the data over may shorten the amount of time for the new member to become current.

  Ensure that you can copy the data directory to the new member and begin replication within the *window allowed by the oplog*. Otherwise, the new instance will have to perform an initial sync, which completely resynchronizes the data, as described in *Resync a Member of a Replica Set* (page 193).

  Use `rs.printReplicationInfo()` to check the current state of replica set members with regards to the oplog.

For background on replication deployment patterns, see the `https://docs.mongodb.org/manual/core/replica-set-ar` document.

Add a Member to an Existing Replica Set

1. Start the new `mongod` instance. Specify the data directory and the replica set name. The following example specifies the `/srv/mongodb/db0` data directory and the `rs0` replica set:

   ```
   mongod --dbpath /srv/mongodb/db0 --replSet rs0
   ```

   Take note of the host name and port information for the new `mongod` instance.

   For more information on configuration options, see the `mongod` manual page.

   Optional

   You can specify the data directory and replica set in the `mongod.conf configuration file`, and start the `mongod` with the following command:

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

2. Connect to the replica set's primary.

   You can only add members while connected to the primary. If you do not know which member is the primary, log into any member of the replica set and issue the `db.isMaster()` command.

3. Use `rs.add()` to add the new member to the replica set. For example, to add a member at host `mongodb3.example.net`, issue the following command:

   ```
   rs.add("mongodb3.example.net")
   ```

   You can include the port number, depending on your setup:

   ```
   rs.add("mongodb3.example.net:27017")
   ```

4. Verify that the member is now part of the replica set. Call the `rs.conf()` method, which displays the `replica set configuration`:

   ```
   rs.conf()
   ```

   To view replica set status, issue the `rs.status()` method. For a description of the status fields, see `https://docs.mongodb.org/manual/reference/command/replSetGetStatus`.

Configure and Add a Member You can add a member to a replica set by passing to the `rs.add()` method a `members` document. The document must be in the form of a `replSetGetConfig.members` document. These documents define a replica set member in the same form as the *replica set configuration document*.

Important: Specify a value for the `_id` field of the `members` document. MongoDB does not automatically populate the `_id` field in this case. Finally, the `members` document must declare the `host` value. All other fields are optional.

Example

To add a member with the following configuration:

- an `_id` of `1`.
- a `hostname and port number` of `mongodb3.example.net:27017`.
- a `priority` value within the replica set of `0`.
- a configuration as `hidden`,

Issue the following:

```
rs.add({_id: 1, host: "mongodb3.example.net:27017", priority: 0, hidden: true})
```

Remove Members from Replica Set

On this page

To remove a member of a *replica set* use either of the following procedures.

Remove a Member Using `rs.remove()`

1. Shut down the `mongod` instance for the member you wish to remove. To shut down the instance, connect using the `mongo` shell and the `db.shutdownServer()` method.
2. Connect to the replica set's current *primary*. To determine the current primary, use `db.isMaster()` while connected to any member of the replica set.
3. Use `rs.remove()` in either of the following forms to remove the member:

   ```
   rs.remove("mongod3.example.net:27017")
   rs.remove("mongod3.example.net")
   ```

   MongoDB disconnects the shell briefly as the replica set elects a new primary. The shell then automatically reconnects. The shell displays a `DBClientCursor::init call() failed` error even though the command succeeds.

Remove a Member Using `rs.reconfig()`

To remove a member you can manually edit the `replica set configuration document`, as described here.

1. Shut down the `mongod` instance for the member you wish to remove. To shut down the instance, connect using the `mongo` shell and the `db.shutdownServer()` method.
2. Connect to the replica set's current *primary*. To determine the current primary, use `db.isMaster()` while connected to any member of the replica set.
3. Issue the `rs.conf()` method to view the current configuration document and determine the position in the `members` array of the member to remove:

   Example

   `mongod_C.example.net` is in position `2` of the following configuration file:

   ```
   {
       "_id" : "rs",
       "version" : 7,
       "members" : [
           {
               "_id" : 0,
               "host" : "mongod_A.example.net:27017"
           },
           {
               "_id" : 1,
               "host" : "mongod_B.example.net:27017"
           },
           {
               "_id" : 2,
               "host" : "mongod_C.example.net:27017"
           }
       ]
   }
   ```

4. Assign the current configuration document to the variable `cfg`:

   ```
   cfg = rs.conf()
   ```

5. Modify the `cfg` object to remove the member.

   Example

   To remove `mongod_C.example.net:27017` use the following JavaScript operation:

   ```
   cfg.members.splice(2,1)
   ```

6. Overwrite the replica set configuration document with the new configuration by issuing the following:

   ```
   rs.reconfig(cfg)
   ```

   As a result of `rs.reconfig()` the shell will disconnect while the replica set renegotiates which member is primary. The shell displays a `DBClientCursor::init call() failed` error even though the command succeeds, and will automatically reconnected.

7. To confirm the new configuration, issue `rs.conf()`.

   For the example above the output would be:

   ```
   {
       "_id" : "rs",
       "version" : 8,
       "members" : [
   ```

```
            {
                "_id" : 0,
                "host" : "mongod_A.example.net:27017"
            },
            {
                "_id" : 1,
                "host" : "mongod_B.example.net:27017"
            }
        ]
    }
```

Replace a Replica Set Member

On this page

If you need to change the hostname of a replica set member without changing the configuration of that member or the set, you can use the operation outlined in this tutorial. For example if you must re-provision systems or rename hosts, you can use this pattern to minimize the scope of that change.

Operation

To change the hostname for a replica set member modify the `host` field. The value of `_id` field will not change when you reconfigure the set.

See `https://docs.mongodb.org/manual/reference/replica-configuration` and `rs.reconfig()` for more information.

Note: Any replica set configuration change can trigger the current *primary* to step down, which forces an *election*. During the election, the current shell session and clients connected to this replica set disconnect, which produces an error even when the operation succeeds.

Example

To change the hostname to `mongo2.example.net` for the replica set member configured at `members[0]`, issue the following sequence of commands:

```
cfg = rs.conf()
cfg.members[0].host = "mongo2.example.net"
rs.reconfig(cfg)
```

5.1.2 Member Configuration Tutorials

The following tutorials provide information in configuring replica set members to support specific operations, such as to provide dedicated backups, to support reporting, or to act as a cold standby.

Adjust Priority for Replica Set Member **(page 179)** Change the precedence given to a replica set members in an election for primary.

***Prevent Secondary from Becoming Primary* (page 180)** Make a secondary member ineligible for election as primary.

***Configure a Hidden Replica Set Member* (page 181)** Configure a secondary member to be invisible to applications in order to support significantly different usage, such as a dedicated backups.

***Configure a Delayed Replica Set Member* (page 183)** Configure a secondary member to keep a delayed copy of the data set in order to provide a rolling backup.

***Configure Non-Voting Replica Set Member* (page 184)** Create a secondary member that keeps a copy of the data set but does not vote in an election.

***Convert a Secondary to an Arbiter* (page 185)** Convert a secondary to an arbiter.

Adjust Priority for Replica Set Member

On this page

Overview

The priority settings of replica set members affect the outcomes of `elections` for primary. Use this setting to ensure that some members are more likely to become primary and that others can never become primary.

The value of the member's `priority` setting determines the member's priority in elections. The higher the number, the higher the priority.

Considerations

To modify priorities, you update the `members` array in the replica configuration object. The array index begins with `0`. Do **not** confuse this index value with the value of the replica set member's `_id` field in the array.

The value of `priority` can be any floating point (i.e. decimal) number between `0` and `1000`. The default value for the `priority` field is `1`.

To block a member from seeking election as primary, assign it a priority of `0`. *Hidden members*, *delayed members*, and *arbiters* all have `priority` set to `0`.

Adjust priority during a scheduled maintenance window. Reconfiguring priority can force the current primary to step down, leading to an election. Before an election the primary closes all open *client* connections.

Procedure

Step 1: Copy the replica set configuration to a variable. In the `mongo` shell, use `rs.conf()` to retrieve the replica set configuration and assign it to a variable. For example:

```
cfg = rs.conf()
```

Step 2: Change each member's priority value. Change each member's `priority` value, as configured in the `members` array.

```
cfg.members[0].priority = 0.5
cfg.members[1].priority = 2
cfg.members[2].priority = 2
```

This sequence of operations modifies the value of `cfg` to set the priority for the first three members defined in the `members` array.

Step 3: Assign the replica set the new configuration. Use `rs.reconfig()` to apply the new configuration.

```
rs.reconfig(cfg)
```

This operation updates the configuration of the replica set using the configuration defined by the value of `cfg`.

Prevent Secondary from Becoming Primary

On this page

Overview

In a replica set, by default all *secondary* members are eligible to become primary through the election process. You can use the `priority` to affect the outcome of these elections by making some members more likely to become primary and other members less likely or unable to become primary.

Secondaries that cannot become primary are also unable to trigger elections. In all other respects these secondaries are identical to other secondaries.

To prevent a *secondary* member from ever becoming a *primary* in a *failover*, assign the secondary a priority of `0`, as described here. For a detailed description of secondary-only members and their purposes, see `https://docs.mongodb.org/manual/core/replica-set-priority-0-member`.

Considerations

When updating the replica configuration object, access the replica set members in the `members` array with the **array index**. The array index begins with `0`. Do **not** confuse this index value with the value of the `_id` field in each document in the `members` array.

Note: MongoDB does not permit the current *primary* to have a priority of `0`. To prevent the current primary from again becoming a primary, you must first step down the current primary using `rs.stepDown()`.

Procedure

This tutorial uses a sample replica set with 5 members.

Warning:

- The `rs.reconfig()` shell method can force the current primary to step down, which causes an *election*. When the primary steps down, the `mongod` closes all client connections. While this typically takes 10-20 seconds, try to make these changes during scheduled maintenance periods.
- To successfully reconfigure a replica set, a majority of the members must be accessible. If your replica set has an even number of members, add an *arbiter* (page 171) to ensure that members can quickly obtain a majority of votes in an election for primary.

Step 1: Retrieve the current replica set configuration. The `rs.conf()` method returns a `replica set configuration document` that contains the current configuration for a replica set.

In a `mongo` shell connected to a primary, run the `rs.conf()` method and assign the result to a variable:

```
cfg = rs.conf()
```

The returned document contains a `members` field which contains an array of member configuration documents, one document for each member of the replica set.

Step 2: Assign priority value of 0. To prevent a secondary member from becoming a primary, update the secondary member's `priority` to `0`.

To assign a priority value to a member of the replica set, access the member configuration document using the array index. In this tutorial, the secondary member to change corresponds to the configuration document found at position `2` of the `members` array.

```
cfg.members[2].priority = 0
```

The configuration change does not take effect until you reconfigure the replica set.

Step 3: Reconfigure the replica set. Use `rs.reconfig()` method to reconfigure the replica set with the updated replica set configuration document.

Pass the `cfg` variable to the `rs.reconfig()` method:

```
rs.reconfig(cfg)
```

Related Documents

- `priority`
- *Adjust Priority for Replica Set Member* (page 179)
- *Replica Set Reconfiguration*
- `https://docs.mongodb.org/manual/core/replica-set-elections`

Configure a Hidden Replica Set Member

On this page

Hidden members are part of a *replica set* but cannot become *primary* and are invisible to client applications. Hidden members may vote in *elections*. For a more information on hidden members and their uses, see `https://docs.mongodb.org/manual/core/replica-set-hidden-member`.

Considerations

The most common use of hidden nodes is to support `delayed members`. If you only need to prevent a member from becoming primary, configure a `priority 0 member`.

If the `chainingAllowed` setting allows secondary members to sync from other secondaries, MongoDB by default prefers non-hidden members over hidden members when selecting a sync target. MongoDB will only choose hidden members as a last resort. If you want a secondary to sync from a hidden member, use the `replSetSyncFrom` database command to override the default sync target. See the documentation for `replSetSyncFrom` before using the command.

See also:

Manage Chained Replication (page 201)

Changed in version 2.0: For *sharded clusters* running with replica sets before 2.0, if you reconfigured a member as hidden, you *had* to restart `mongos` to prevent queries from reaching the hidden member.

Examples

Member Configuration Document To configure a secondary member as hidden, set its `priority` value to `0` and set its `hidden` value to `true` in its member configuration:

```
{
  "_id" : <num>
  "host" : <hostname:port>,
  "priority" : 0,
  "hidden" : true
}
```

Configuration Procedure The following example hides the secondary member currently at the index `0` in the `members` array. To configure a *hidden member*, use the following sequence of operations in a `mongo` shell connected to the primary, specifying the member to configure by its array index in the `members` array:

```
cfg = rs.conf()
cfg.members[0].priority = 0
cfg.members[0].hidden = true
rs.reconfig(cfg)
```

After re-configuring the set, this secondary member has a priority of `0` so that it cannot become primary and is hidden. The other members in the set will not advertise the hidden member in the `isMaster` or `db.isMaster()` output.

When updating the replica configuration object, access the replica set members in the `members` array with the **array index**. The array index begins with `0`. Do **not** confuse this index value with the value of the `_id` field in each document in the `members` array.

Warning:

- The `rs.reconfig()` shell method can force the current primary to step down, which causes an *election*. When the primary steps down, the `mongod` closes all client connections. While this typically takes 10-20 seconds, try to make these changes during scheduled maintenance periods.
- To successfully reconfigure a replica set, a majority of the members must be accessible. If your replica set has an even number of members, add an *arbiter* (page 171) to ensure that members can quickly obtain a majority of votes in an election for primary.

Related Documents

- *Replica Set Reconfiguration*
- `https://docs.mongodb.org/manual/core/replica-set-elections`
- *Read Preference*

Configure a Delayed Replica Set Member

On this page

To configure a delayed secondary member, set its `priority` value to `0`, its `hidden` value to `true`, and its `slaveDelay` value to the number of seconds to delay.

Important: The length of the secondary `slaveDelay` must fit within the window of the oplog. If the oplog is shorter than the `slaveDelay` window, the delayed member cannot successfully replicate operations.

When you configure a delayed member, the delay applies both to replication and to the member's *oplog*. For details on delayed members and their uses, see `https://docs.mongodb.org/manual/core/replica-set-delayed-member`.

Example

The following example sets a 1-hour delay on a secondary member currently at the index `0` in the `members` array. To set the delay, issue the following sequence of operations in a `mongo` shell connected to the primary:

```
cfg = rs.conf()
cfg.members[0].priority = 0
cfg.members[0].hidden = true
cfg.members[0].slaveDelay = 3600
rs.reconfig(cfg)
```

After the replica set reconfigures, the delayed secondary member cannot become *primary* and is hidden from applications. The `slaveDelay` value delays both replication and the member's *oplog* by 3600 seconds (1 hour).

When updating the replica configuration object, access the replica set members in the `members` array with the **array index**. The array index begins with `0`. Do **not** confuse this index value with the value of the `_id` field in each document in the `members` array.

Warning:

- The `rs.reconfig()` shell method can force the current primary to step down, which causes an *election*. When the primary steps down, the `mongod` closes all client connections. While this typically takes 10-20 seconds, try to make these changes during scheduled maintenance periods.
- To successfully reconfigure a replica set, a majority of the members must be accessible. If your replica set has an even number of members, add an *arbiter* (page 171) to ensure that members can quickly obtain a majority of votes in an election for primary.

Related Documents

- `slaveDelay`
- *Replica Set Reconfiguration*
- *replica-set-oplog-sizing*
- *Change the Size of the Oplog* (page 187) tutorial
- `https://docs.mongodb.org/manual/core/replica-set-elections`

Configure Non-Voting Replica Set Member

On this page

Non-voting members allow you to add additional members for read distribution beyond the maximum seven voting members. To configure a member as non-voting, set its `votes` value to `0`.

Example

To disable the ability to vote in elections for the fourth, fifth, and sixth replica set members, use the following command sequence in the `mongo` shell connected to the primary. You identify each replica set member by its array index in the `members` array:

```
cfg = rs.conf()
cfg.members[3].votes = 0
cfg.members[4].votes = 0
cfg.members[5].votes = 0
rs.reconfig(cfg)
```

This sequence gives `0` votes to the fourth, fifth, and sixth members of the set according to the order of the `members` array in the output of `rs.conf()`. This setting allows the set to elect these members as *primary* but does not allow them to vote in elections. Place voting members so that your designated primary or primaries can reach a majority of votes in the event of a network partition.

When updating the replica configuration object, access the replica set members in the `members` array with the **array index**. The array index begins with `0`. Do **not** confuse this index value with the value of the `_id` field in each document in the `members` array.

Warning:

- The `rs.reconfig()` shell method can force the current primary to step down, which causes an *election*. When the primary steps down, the `mongod` closes all client connections. While this typically takes 10-20 seconds, try to make these changes during scheduled maintenance periods.
- To successfully reconfigure a replica set, a majority of the members must be accessible. If your replica set has an even number of members, add an *arbiter* (page 171) to ensure that members can quickly obtain a majority of votes in an election for primary.

In general and when possible, all members should have only 1 vote. This prevents intermittent ties, deadlocks, or the wrong members from becoming primary. Use `priority` to control which members are more likely to become primary.

Related Documents

- `votes`
- *Replica Set Reconfiguration*
- `https://docs.mongodb.org/manual/core/replica-set-elections`

Convert a Secondary to an Arbiter

On this page

If you have a *secondary* in a *replica set* that no longer needs to hold data but that needs to remain in the set to ensure that the set can *elect a primary*, you may convert the secondary to an *arbiter* using either procedure in this tutorial. Both procedures are operationally equivalent:

- You may operate the arbiter on the same port as the former secondary. In this procedure, you must shut down the secondary and remove its data before restarting and reconfiguring it as an arbiter.

  For this procedure, see *Convert Secondary to Arbiter and Reuse the Port Number* (page 185).

- Run the arbiter on a new port. In this procedure, you can reconfigure the server as an arbiter before shutting down the instance running as a secondary.

  For this procedure, see *Convert Secondary to Arbiter Running on a New Port Number* (page 186).

Convert Secondary to Arbiter and Reuse the Port Number

1. If your application is connecting directly to the secondary, modify the application so that MongoDB queries don't reach the secondary.
2. Shut down the secondary.
3. Remove the *secondary* from the *replica set* by calling the `rs.remove()` method. Perform this operation while connected to the current *primary* in the `mongo` shell:

   ```
   rs.remove("<hostname><:port>")
   ```

4. Verify that the replica set no longer includes the secondary by calling the `rs.conf()` method in the `mongo` shell:

   ```
   rs.conf()
   ```

5. Move the secondary's data directory to an archive folder. For example:

   ```
   mv /data/db /data/db-old
   ```

   Optional

   You may remove the data instead.

6. Create a new, empty data directory to point to when restarting the `mongod` instance. You can reuse the previous name. For example:

   ```
   mkdir /data/db
   ```

7. Restart the `mongod` instance for the secondary, specifying the port number, the empty data directory, and the replica set. You can use the same port number you used before. Issue a command similar to the following:

   ```
   mongod --port 27021 --dbpath /data/db --replSet rs
   ```

8. In the `mongo` shell convert the secondary to an arbiter using the `rs.addArb()` method:

   ```
   rs.addArb("<hostname><:port>")
   ```

9. Verify the arbiter belongs to the replica set by calling the `rs.conf()` method in the `mongo` shell.

   ```
   rs.conf()
   ```

   The arbiter member should include the following:

   ```
   "arbiterOnly" : true
   ```

Convert Secondary to Arbiter Running on a New Port Number

1. If your application is connecting directly to the secondary or has a connection string referencing the secondary, modify the application so that MongoDB queries don't reach the secondary.
2. Create a new, empty data directory to be used with the new port number. For example:

   ```
   mkdir /data/db-temp
   ```

3. Start a new `mongod` instance on the new port number, specifying the new data directory and the existing replica set. Issue a command similar to the following:

   ```
   mongod --port 27021 --dbpath /data/db-temp --replSet rs
   ```

4. In the `mongo` shell connected to the current primary, convert the new `mongod` instance to an arbiter using the `rs.addArb()` method:

   ```
   rs.addArb("<hostname><:port>")
   ```

5. Verify the arbiter has been added to the replica set by calling the `rs.conf()` method in the `mongo` shell.

   ```
   rs.conf()
   ```

   The arbiter member should include the following:

```
"arbiterOnly" : true
```

6. Shut down the secondary.
7. Remove the *secondary* from the *replica set* by calling the `rs.remove()` method in the `mongo` shell:

   ```
   rs.remove("<hostname><:port>")
   ```

8. Verify that the replica set no longer includes the old secondary by calling the `rs.conf()` method in the `mongo` shell:

   ```
   rs.conf()
   ```

9. Move the secondary's data directory to an archive folder. For example:

   ```
   mv /data/db /data/db-old
   ```

   Optional

   You may remove the data instead.

5.1.3 Replica Set Maintenance Tutorials

The following tutorials provide information in maintaining existing replica sets.

Change the Size of the Oplog **(page 187)** Increase the size of the *oplog* which logs operations. In most cases, the default oplog size is sufficient.

Perform Maintenance on Replica Set Members **(page 190)** Perform maintenance on a member of a replica set while minimizing downtime.

Force a Member to Become Primary **(page 191)** Force a replica set member to become primary.

Resync a Member of a Replica Set **(page 193)** Sync the data on a member. Either perform initial sync on a new member or resync the data on an existing member that has fallen too far behind to catch up by way of normal replication.

Configure Replica Set Tag Sets **(page 194)** Assign tags to replica set members for use in targeting read and write operations to specific members.

Reconfigure a Replica Set with Unavailable Members **(page 198)** Reconfigure a replica set when a majority of replica set members are down or unreachable.

Manage Chained Replication **(page 201)** Disable or enable chained replication. Chained replication occurs when a secondary replicates from another secondary instead of the primary.

Change Hostnames in a Replica Set **(page 202)** Update the replica set configuration to reflect changes in members' hostnames.

Configure a Secondary's Sync Target **(page 206)** Specify the member that a secondary member synchronizes from.

Change the Size of the Oplog

On this page

The *oplog* exists internally as a *capped collection*, so you cannot modify its size in the course of normal operations. In most cases the *default oplog size* is an acceptable size; however, in some situations you may need a larger or smaller oplog. For example, you might need to change the oplog size if your applications perform large numbers of multi-updates or deletes in short periods of time.

This tutorial describes how to resize the oplog. For a detailed explanation of oplog sizing, see *replica-set-oplog-sizing*. For details how oplog size affects *delayed members* and affects *replication lag*, see *replica-set-delayed-members*.

Overview

To change the size of the oplog, you must perform maintenance on each member of the replica set in turn. The procedure requires: stopping the `mongod` instance and starting as a standalone instance, modifying the oplog size, and restarting the member.

Important: Always start rolling replica set maintenance with the secondaries, and finish with the maintenance on primary member.

Procedure

- Restart the member in standalone mode.

  Tip

  Always use `rs.stepDown()` to force the primary to become a secondary, before stopping the server. This facilitates a more efficient election process.

- Recreate the oplog with the new size and with an old oplog entry as a seed.
- Restart the `mongod` instance as a member of the replica set.

Restart a Secondary in Standalone Mode on a Different Port Shut down the `mongod` instance for one of the non-primary members of your replica set. For example, to shut down, use the `db.shutdownServer()` method:

```
db.shutdownServer()
```

Restart this `mongod` as a standalone instance running on a different port and *without* the `--replSet` parameter. Use a command similar to the following:

```
mongod --port 37017 --dbpath /srv/mongodb
```

Create a Backup of the Oplog (Optional) Optionally, backup the existing oplog on the standalone instance, as in the following example:

```
mongodump --db local --collection 'oplog.rs' --port 37017
```

Recreate the Oplog with a New Size and a Seed Entry Save the last entry from the oplog. For example, connect to the instance using the `mongo` shell, and enter the following command to switch to the `local` database:

```
use local
```

In `mongo` shell scripts you can use the following operation to set the `db` object:

```
db = db.getSiblingDB('local')
```

Ensure that the `temp` temporary collection is empty by dropping the collection:

```
db.temp.drop()
```

Use the `db.collection.save()` method and a sort on reverse *natural order* to find the last entry and save it to a temporary collection:

```
db.temp.save( db.oplog.rs.find( { }, { ts: 1, h: 1 } ).sort( {$natural : -1} ).limit(1).next() )
```

To see this oplog entry, use the following operation:

```
db.temp.find()
```

Remove the Existing Oplog Collection Drop the old `oplog.rs` collection in the `local` database. Use the following command:

```
db = db.getSiblingDB('local')
db.oplog.rs.drop()
```

This returns `true` in the shell.

Create a New Oplog Use the `create` command to create a new oplog of a different size. Specify the `size` argument in bytes. A value of `2 * 1024 * 1024 * 1024` will create a new oplog that's 2 gigabytes:

```
db.runCommand( { create: "oplog.rs", capped: true, size: (2 * 1024 * 1024 * 1024) } )
```

Upon success, this command returns the following status:

```
{ "ok" : 1 }
```

Insert the Last Entry of the Old Oplog into the New Oplog Insert the previously saved last entry from the old oplog into the new oplog. For example:

```
db.oplog.rs.save( db.temp.findOne() )
```

To confirm the entry is in the new oplog, use the following operation:

```
db.oplog.rs.find()
```

Restart the Member Restart the `mongod` as a member of the replica set on its usual port. For example:

```
db.shutdownServer()
mongod --replSet rs0 --dbpath /srv/mongodb
```

The replica set member will recover and "catch up" before it is eligible for election to primary.

Repeat Process for all Members that may become Primary Repeat this procedure for all members you want to change the size of the oplog. Repeat the procedure for the primary as part of the following step.

Change the Size of the Oplog on the Primary To finish the rolling maintenance operation, step down the primary with the `rs.stepDown()` method and repeat the oplog resizing procedure above.

Perform Maintenance on Replica Set Members

On this page

Overview

Replica sets allow a MongoDB deployment to remain available during the majority of a maintenance window.

This document outlines the basic procedure for performing maintenance on each of the members of a replica set. Furthermore, this particular sequence strives to minimize the amount of time that the *primary* is unavailable and controlling the impact on the entire deployment.

Use these steps as the basis for common replica set operations, particularly for procedures such as *upgrading to the latest version of MongoDB* (page 66) and *changing the size of the oplog* (page 187).

Procedure

For each member of a replica set, starting with a secondary member, perform the following sequence of events, ending with the primary:

- Restart the `mongod` instance as a standalone.
- Perform the task on the standalone instance.
- Restart the `mongod` instance as a member of the replica set.

Step 1: Stop a secondary. In the `mongo` shell, shut down the `mongod` instance:

```
db.shutdownServer()
```

Step 2: Restart the secondary as a standalone on a different port. At the operating system shell prompt, restart `mongod` as a standalone instance running on a different port and *without* the `--replSet` parameter:

```
mongod --port 37017 --dbpath /srv/mongodb
```

Always start `mongod` with the same user, even when restarting a replica set member as a standalone instance.

Step 3: Perform maintenance operations on the secondary. While the member is a standalone, use the `mongo` shell to perform maintenance:

```
mongo --port 37017
```

Step 4: Restart `mongod` as a member of the replica set. After performing all maintenance tasks, use the following procedure to restart the `mongod` as a member of the replica set on its usual port.

From the `mongo` shell, shut down the standalone server after completing the maintenance:

```
db.shutdownServer()
```

Restart the `mongod` instance as a member of the replica set using its normal command-line arguments or configuration file.

The secondary takes time to `catch up to the primary`. From the `mongo` shell, use the following command to verify that the member has caught up from the `RECOVERING` state to the `SECONDARY` state.

```
rs.status()
```

Step 5: Perform maintenance on the primary last. To perform maintenance on the primary after completing maintenance tasks on all secondaries, use `rs.stepDown()` in the `mongo` shell to step down the primary and allow one of the secondaries to be elected the new primary. Specify a 300 second waiting period to prevent the member from being elected primary again for five minutes:

```
rs.stepDown(300)
```

After the primary steps down, the replica set will elect a new primary. See `https://docs.mongodb.org/manual/core/replica-set-elections` for more information about replica set elections.

Force a Member to Become Primary

On this page

Overview

You can force a *replica set* member to become *primary* by giving it a higher `priority` value than any other member in the set.

Optionally, you also can force a member never to become primary by setting its `priority` value to `0`, which means the member can never seek *election* as primary. For more information, see *replica-set-secondary-only-members*.

For more information on priorities, see `priority`.

Consideration

A majority of the configured members of a replica set *must* be available for a set to reconfigure a set or elect a primary. See `https://docs.mongodb.org/manual/core/replica-set-elections` for more information.

Procedures

Force a Member to be Primary by Setting its Priority High This procedure assumes your current *primary* is `m1.example.net` and that you'd like to instead make `m3.example.net` primary. The procedure also assumes you have a three-member *replica set* with the configuration below. For more information on configurations, see *Replica Set Configuration Use*.

This procedure assumes this configuration:

```
{
    "_id" : "rs",
    "version" : 7,
    "members" : [
        {
            "_id" : 0,
            "host" : "m1.example.net:27017"
        },
        {
            "_id" : 1,
            "host" : "m2.example.net:27017"
        },
        {
            "_id" : 2,
            "host" : "m3.example.net:27017"
        }
    ]
}
```

1. In a `mongo` shell connected to the primary, use the following sequence of operations to make `m3.example.net` the primary:

   ```
   cfg = rs.conf()
   cfg.members[0].priority = 0.5
   cfg.members[1].priority = 0.5
   cfg.members[2].priority = 1
   rs.reconfig(cfg)
   ```

   The last statement calls `rs.reconfig()` with the modified configuration document to configure `m3.example.net` to have a higher `replSetGetConfig.members[n].priority` value than the other `mongod` instances.

   The following sequence of events occur:

   - `m3.example.net` and `m2.example.net` sync with `m1.example.net` (typically within 10 seconds).
   - `m1.example.net` sees that it no longer has highest priority and, in most cases, steps down. `m1.example.net` *does not* step down if `m3.example.net`'s sync is far behind. In that case, `m1.example.net` waits until `m3.example.net` is within 10 seconds of its optime and then steps down. This minimizes the amount of time with no primary following failover.
   - The step down forces on election in which `m3.example.net` becomes primary based on its `priority` setting.

2. Optionally, if `m3.example.net` is more than 10 seconds behind `m1.example.net`'s optime, and if you don't need to have a primary designated within 10 seconds, you can force `m1.example.net` to step down by running:

   ```
   db.adminCommand({replSetStepDown: 86400, force: 1})
   ```

   This prevents `m1.example.net` from being primary for 86,400 seconds (24 hours), even if there is no other member that can become primary. When `m3.example.net` catches up with `m1.example.net` it will become primary.

   If you later want to make `m1.example.net` primary again while it waits for `m3.example.net` to catch up, issue the following command to make `m1.example.net` seek election again:

   ```
   rs.freeze()
   ```

The `rs.freeze()` provides a wrapper around the `replSetFreeze` database command.

Force a Member to be Primary Using Database Commands Changed in version 1.8.

Consider a *replica set* with the following members:

- `mdb0.example.net` - the current *primary*.
- `mdb1.example.net` - a *secondary*.
- `mdb2.example.net` - a secondary .

To force a member to become primary use the following procedure:

1. In a `mongo` shell, run `rs.status()` to ensure your replica set is running as expected.
2. In a `mongo` shell connected to the `mongod` instance running on `mdb2.example.net`, freeze `mdb2.example.net` so that it does not attempt to become primary for 120 seconds.

   ```
   rs.freeze(120)
   ```

3. In a `mongo` shell connected the `mongod` running on `mdb0.example.net`, step down this instance that the `mongod` is not eligible to become primary for 120 seconds:

   ```
   rs.stepDown(120)
   ```

   `mdb1.example.net` becomes primary.

   Note: During the transition, there is a short window where the set does not have a primary.

For more information, consider the `rs.freeze()` and `rs.stepDown()` methods that wrap the `replSetFreeze` and `replSetStepDown` commands.

Resync a Member of a Replica Set

On this page

A *replica set* member becomes "stale" when its replication process falls so far behind that the *primary* overwrites oplog entries the member has not yet replicated. The member cannot catch up and becomes "stale." When this occurs, you must completely resynchronize the member by removing its data and performing an *initial sync*.

This tutorial addresses both resyncing a stale member and to creating a new member using seed data from another member. When syncing a member, choose a time when the system has the bandwidth to move a large amount of data. Schedule the synchronization during a time of low usage or during a maintenance window.

MongoDB provides two options for performing an initial sync:

- Restart the `mongod` with an empty data directory and let MongoDB's normal initial syncing feature restore the data. This is the more simple option but may take longer to replace the data.

  See *Procedures* (page 194).

- Restart the machine with a copy of a recent data directory from another member in the replica set. This procedure can replace the data more quickly but requires more manual steps.

  See *Sync by Copying Data Files from Another Member* (page 194).

Procedures

Automatically Sync a Member

Warning: During initial sync, `mongod` will remove the content of the `dbPath`.

This procedure relies on MongoDB's regular process for *initial sync*. This will store the current data on the member. For an overview of MongoDB initial sync process, see the *replica-set-syncing* section.

If the instance has no data, you can simply follow the *Add Members to a Replica Set* (page 173) or *Replace a Replica Set Member* (page 178) procedure to add a new member to a replica set.

You can also force a `mongod` that is already a member of the set to to perform an initial sync by restarting the instance without the content of the `dbPath` as follows:

1. Stop the member's `mongod` instance. To ensure a clean shutdown, use the `db.shutdownServer()` method from the `mongo` shell or on Linux systems, the *`mongod --shutdown`* option.
2. Delete all data and sub-directories from the member's data directory. By removing the data `dbPath`, MongoDB will perform a complete resync. Consider making a backup first.

At this point, the `mongod` will perform an initial sync. The length of the initial sync process depends on the size of the database and network connection between members of the replica set.

Initial sync operations can impact the other members of the set and create additional traffic to the primary and can only occur if another member of the set is accessible and up to date.

Sync by Copying Data Files from Another Member This approach "seeds" a new or stale member using the data files from an existing member of the replica set. The data files **must** be sufficiently recent to allow the new member to catch up with the *oplog*. Otherwise the member would need to perform an initial sync.

Copy the Data Files You can capture the data files as either a snapshot or a direct copy. However, in most cases you cannot copy data files from a running `mongod` instance to another because the data files will change during the file copy operation.

Important: If copying data files, you must copy the content of the `local` database.

You *cannot* use a `mongodump` backup for the data files, **only a snapshot backup**. For approaches to capturing a consistent snapshot of a running `mongod` instance, see the *MongoDB Backup Methods* (page 4) documentation.

Sync the Member After you have copied the data files from the "seed" source, start the `mongod` instance and allow it to apply all operations from the oplog until it reflects the current state of the replica set.

Configure Replica Set Tag Sets

On this page

Tag sets let you customize *write concern* and *read preferences* for a *replica set*. MongoDB stores tag sets in the replica set configuration object, which is the document returned by `rs.conf()`, in the `members[n].tags` embedded document.

This section introduces the configuration of tag sets. For an overview on tag sets and their use, see *Replica Set Write Concern* and *replica-set-read-preference-tag-sets*.

Differences Between Read Preferences and Write Concerns

Custom read preferences and write concerns evaluate tags sets in different ways:

- Read preferences consider the value of a tag when selecting a member to read from.
- Write concerns do not use the value of a tag to select a member except to consider whether or not the value is unique.

For example, a tag set for a read operation may resemble the following document:

```
{ "disk": "ssd", "use": "reporting" }
```

To fulfill such a read operation, a member would need to have both of these tags. Any of the following tag sets would satisfy this requirement:

```
{ "disk": "ssd", "use": "reporting" }
{ "disk": "ssd", "use": "reporting", "rack": "a" }
{ "disk": "ssd", "use": "reporting", "rack": "d" }
{ "disk": "ssd", "use": "reporting", "mem": "r"}
```

The following tag sets would *not* be able to fulfill this query:

```
{ "disk": "ssd" }
{ "use": "reporting" }
{ "disk": "ssd", "use": "production" }
{ "disk": "ssd", "use": "production", "rack": "k" }
{ "disk": "spinning", "use": "reporting", "mem": "32" }
```

Add Tag Sets to a Replica Set

Given the following replica set configuration:

```
{
    "_id" : "rs0",
    "version" : 1,
    "members" : [
              {
                      "_id" : 0,
                      "host" : "mongodb0.example.net:27017"
              },
              {
                      "_id" : 1,
                      "host" : "mongodb1.example.net:27017"
              },
              {
                      "_id" : 2,
                      "host" : "mongodb2.example.net:27017"
              }
     ]
}
```

You could add tag sets to the members of this replica set with the following command sequence in the `mongo` shell:

```
conf = rs.conf()
conf.members[0].tags = { "dc": "east", "use": "production"  }
conf.members[1].tags = { "dc": "east", "use": "reporting"  }
conf.members[2].tags = { "use": "production"  }
rs.reconfig(conf)
```

After this operation the output of `rs.conf()` would resemble the following:

```
{
    "_id" : "rs0",
    "version" : 2,
    "members" : [
             {
                     "_id" : 0,
                     "host" : "mongodb0.example.net:27017",
                     "tags" : {
                             "dc": "east",
                             "use": "production"
                     }
             },
             {
                     "_id" : 1,
                     "host" : "mongodb1.example.net:27017",
                     "tags" : {
                             "dc": "east",
                             "use": "reporting"
                     }
             },
             {
                     "_id" : 2,
                     "host" : "mongodb2.example.net:27017",
                     "tags" : {
                             "use": "production"
                     }
             }
     ]
}
```

Important: In tag sets, all tag values must be strings.

Custom Multi-Datacenter Write Concerns

Given a five member replica set with members in two data centers:

1. a facility `VA` tagged `dc_va`
2. a facility `GTO` tagged `dc_gto`

Create a custom write concern to require confirmation from two data centers using replica set tags, using the following sequence of operations in the `mongo` shell:

1. Create a replica set configuration JavaScript object `conf`:

   ```
   conf = rs.conf()
   ```

2. Add tags to the replica set members reflecting their locations:

```
conf.members[0].tags = { "dc_va": "rack1"}
conf.members[1].tags = { "dc_va": "rack2"}
conf.members[2].tags = { "dc_gto": "rack1"}
conf.members[3].tags = { "dc_gto": "rack2"}
conf.members[4].tags = { "dc_va": "rack1"}
rs.reconfig(conf)
```

3. Create a custom `getLastErrorModes` setting to ensure that the write operation will propagate to at least one member of each facility:

   ```
   conf.settings = { getLastErrorModes: { MultipleDC : { "dc_va": 1, "dc_gto": 1 } } }
   ```

4. Reconfigure the replica set using the modified `conf` configuration object:

   ```
   rs.reconfig(conf)
   ```

To ensure that a write operation propagates to at least one member of the set in both data centers, use the `MultipleDC` write concern mode as follows:

```
db.users.insert( { id: "xyz", status: "A" }, { writeConcern: { w: "MultipleDC" } } )
```

Alternatively, if you want to ensure that each write operation propagates to at least 2 racks in each facility, reconfigure the replica set as follows in the `mongo` shell:

1. Create a replica set configuration object `conf`:

   ```
   conf = rs.conf()
   ```

2. Redefine the `getLastErrorModes` value to require two different values of both `dc_va` and `dc_gto`:

   ```
   conf.settings = { getLastErrorModes: { MultipleDC : { "dc_va": 2, "dc_gto": 2}}
   ```

3. Reconfigure the replica set using the modified `conf` configuration object:

   ```
   rs.reconfig(conf)
   ```

Now, the following write operation will only return after the write operation propagates to at least two different racks in the each facility:

Changed in version 2.6: A new protocol for *write operations* integrates write concerns with the write operations. Previous versions used the `getLastError` command to specify the write concerns.

```
db.users.insert( { id: "xyz", status: "A" }, { writeConcern: { w: "MultipleDC" } } )
```

Configure Tag Sets for Functional Segregation of Read and Write Operations

Given a replica set with tag sets that reflect:

- data center facility,
- physical rack location of instance, and
- storage system (i.e. disk) type.

Where each member of the set has a tag set that resembles one of the following: [2]

[2] Since read preferences and write concerns use the value of fields in tag sets differently, larger deployments may have some redundancy.

```
{"dc_va": "rack1", disk:"ssd", ssd: "installed" }
{"dc_va": "rack2", disk:"raid"}
{"dc_gto": "rack1", disk:"ssd", ssd: "installed" }
{"dc_gto": "rack2", disk:"raid"}
{"dc_va": "rack1", disk:"ssd", ssd: "installed" }
```

To target a read operation to a member of the replica set with a disk type of `ssd`, you could use the following tag set:

```
{ disk: "ssd" }
```

However, to create comparable write concern modes, you would specify a different set of `getLastErrorModes` configuration. Consider the following sequence of operations in the `mongo` shell:

1. Create a replica set configuration object `conf`:

   ```
   conf = rs.conf()
   ```

2. Redefine the `getLastErrorModes` value to configure two write concern modes:

   ```
   conf.settings = {
                    "getLastErrorModes" : {
                            "ssd" : {
                                       "ssd" : 1
                            },
                            "MultipleDC" : {
                                    "dc_va" : 1,
                                   "dc_gto" : 1
                            }
                    }
                  }
   ```

3. Reconfigure the replica set using the modified `conf` configuration object:

   ```
   rs.reconfig(conf)
   ```

Now you can specify the `MultipleDC` write concern mode, as in the following, to ensure that a write operation propagates to each data center.

Changed in version 2.6: A new protocol for *write operations* integrates write concerns with the write operations. Previous versions used the `getLastError` command to specify the write concerns.

```
db.users.insert( { id: "xyz", status: "A" }, { writeConcern: { w: "MultipleDC" } } )
```

Additionally, you can specify the `ssd` write concern mode to ensure that a write operation propagates to at least one instance with an SSD.

Reconfigure a Replica Set with Unavailable Members

On this page

To reconfigure a *replica set* when a **majority** of members are available, use the `rs.reconfig()` operation on the current *primary*, following the example in the *Replica Set Reconfiguration Procedure*.

This document provides the following options for re-configuring a replica set when *only* a **minority** of members are accessible:

- *Reconfigure by Forcing the Reconfiguration* (page 199)
- *Reconfigure by Replacing the Replica Set* (page 199)

You may need to use one of these procedures, for example, in a geographically distributed replica set, where *no* local group of members can reach a majority. See *replica-set-elections* for more information on this situation.

Reconfigure by Forcing the Reconfiguration

Changed in version 2.0.

This procedure lets you recover while a majority of *replica set* members are down or unreachable. You connect to any surviving member and use the `force` option to the `rs.reconfig()` method.

The `force` option forces a new configuration onto the member. Use this procedure only to recover from catastrophic interruptions. Do not use `force` every time you reconfigure. Also, do not use the `force` option in any automatic scripts and do not use `force` when there is still a *primary*.

To force reconfiguration:

1. Back up a surviving member.
2. Connect to a surviving member and save the current configuration. Consider the following example commands for saving the configuration:

   ```
   cfg = rs.conf()

   printjson(cfg)
   ```

3. On the same member, remove the down and unreachable members of the replica set from the `members` array by setting the array equal to the surviving members alone. Consider the following example, which uses the `cfg` variable created in the previous step:

   ```
   cfg.members = [cfg.members[0] , cfg.members[4] , cfg.members[7]]
   ```

4. On the same member, reconfigure the set by using the `rs.reconfig()` command with the `force` option set to `true`:

   ```
   rs.reconfig(cfg, {force : true})
   ```

   This operation forces the secondary to use the new configuration. The configuration is then propagated to all the surviving members listed in the `members` array. The replica set then elects a new primary.

   Note: When you use `force : true`, the version number in the replica set configuration increases significantly, by tens or hundreds of thousands. This is normal and designed to prevent set version collisions if you accidentally force re-configurations on both sides of a network partition and then the network partitioning ends.

5. If the failure or partition was only temporary, shut down or decommission the removed members as soon as possible.

Reconfigure by Replacing the Replica Set

Use the following procedure **only** for versions of MongoDB prior to version 2.0. If you're running MongoDB 2.0 or later, use the above procedure, *Reconfigure by Forcing the Reconfiguration* (page 199).

These procedures are for situations where a *majority* of the *replica set* members are down or unreachable. If a majority is *running*, then skip these procedures and instead use the `rs.reconfig()` command according to the examples in *replica-set-reconfiguration-usage*.

If you run a pre-2.0 version and a majority of your replica set is down, you have the two options described here. Both involve replacing the replica set.

Reconfigure by Turning Off Replication This option replaces the *replica set* with a *standalone* server.

1. Stop the surviving `mongod` instances. To ensure a clean shutdown, use an existing *init script* or use the `db.shutdownServer()` method.

   For example, to use the `db.shutdownServer()` method, connect to the server using the `mongo` shell and issue the following sequence of commands:

   ```
   use admin
   db.shutdownServer()
   ```

2. Create a backup of the data directory (i.e. `dbPath`) of the surviving members of the set.

   Optional

   If you have a backup of the database you may instead remove this data.

3. Restart one of the `mongod` instances *without* the `--replSet` parameter.

   The data is now accessible and provided by a single server that is not a replica set member. Clients can use this server for both reads and writes.

When possible, re-deploy a replica set to provide redundancy and to protect your deployment from operational interruption.

Reconfigure by "Breaking the Mirror" This option selects a surviving *replica set* member to be the new *primary* and to "seed" a new replica set. In the following procedure, the new primary is `db0.example.net`. MongoDB copies the data from `db0.example.net` to all the other members.

1. Stop the surviving `mongod` instances. To ensure a clean shutdown, use an existing *init script* or use the `db.shutdownServer()` method.

   For example, to use the `db.shutdownServer()` method, connect to the server using the `mongo` shell and issue the following sequence of commands:

   ```
   use admin
   db.shutdownServer()
   ```

2. Move the data directories (i.e. `dbPath`) for all the members except `db0.example.net`, so that all the members except `db0.example.net` have empty data directories. For example:

   ```
   mv /data/db /data/db-old
   ```

3. Move the data files for `local` database (i.e. `local.*`) so that `db0.example.net` has no local database. For example

   ```
   mkdir /data/local-old
   mv /data/db/local* /data/local-old/
   ```

4. Start each member of the replica set normally.

5. Connect to `db0.example.net` in a `mongo` shell and run `rs.initiate()` to initiate the replica set.

6. Add the other set members using `rs.add()`. For example, to add a member running on `db1.example.net` at port `27017`, issue the following command:

```
rs.add("db1.example.net:27017")
```

MongoDB performs an initial sync on the added members by copying all data from `db0.example.net` to the added members.

See also:

Resync a Member of a Replica Set (page 193)

Manage Chained Replication

On this page

Starting in version 2.0, MongoDB supports chained replication. A chained replication occurs when a *secondary* member replicates from another secondary member instead of from the *primary*. This might be the case, for example, if a secondary selects its replication target based on ping time and if the closest member is another secondary.

Chained replication can reduce load on the primary. But chained replication can also result in increased replication lag, depending on the topology of the network.

New in version 2.2.2.

You can use the `chainingAllowed` setting in `https://docs.mongodb.org/manual/reference/replica-configur`
to disable chained replication for situations where chained replication is causing lag.

MongoDB enables chained replication by default. This procedure describes how to disable it and how to re-enable it.

Note: If chained replication is disabled, you still can use `replSetSyncFrom` to specify that a secondary replicates from another secondary. But that configuration will last only until the secondary recalculates which member to sync from.

Disable Chained Replication

To disable chained replication, set the `chainingAllowed` field in `https://docs.mongodb.org/manual/reference/rep`
to `false`.

You can use the following sequence of commands to set `chainingAllowed` to `false`:

1. Copy the configuration settings into the `cfg` object:

   ```
   cfg = rs.config()
   ```

2. Take note of whether the current configuration settings contain the `settings` embedded document. If they do, skip this step.

   Warning: To avoid data loss, skip this step if the configuration settings contain the `settings` embedded document.

   If the current configuration settings **do not** contain the `settings` embedded document, create the embedded document by issuing the following command:

```
cfg.settings = { }
```

3. Issue the following sequence of commands to set `chainingAllowed` to `false`:

```
cfg.settings.chainingAllowed = false
rs.reconfig(cfg)
```

Re-enable Chained Replication

To re-enable chained replication, set `chainingAllowed` to `true`. You can use the following sequence of commands:

```
cfg = rs.config()
cfg.settings.chainingAllowed = true
rs.reconfig(cfg)
```

Change Hostnames in a Replica Set

On this page

For most *replica sets*, the hostnames in the `host` field never change. However, if organizational needs change, you might need to migrate some or all host names.

Note: Always use resolvable hostnames for the value of the `host` field in the replica set configuration to avoid confusion and complexity.

Overview

This document provides two separate procedures for changing the hostnames in the `host` field. Use either of the following approaches:

- *Change hostnames without disrupting availability* (page 203). This approach ensures your applications will always be able to read and write data to the replica set, but the approach can take a long time and may incur downtime at the application layer.

  If you use the first procedure, you must configure your applications to connect to the replica set at both the old and new locations, which often requires a restart and reconfiguration at the application layer and which may affect the availability of your applications. Re-configuring applications is beyond the scope of this document.

- *Stop all members running on the old hostnames at once* (page 204). This approach has a shorter maintenance window, but the replica set will be unavailable during the operation.

See also:

Replica Set Reconfiguration Process, *Deploy a Replica Set* (page 160), and *Add Members to a Replica Set* (page 173).

Assumptions

Given a *replica set* with three members:

- `database0.example.com:27017` (the *primary*)
- `database1.example.com:27017`
- `database2.example.com:27017`

And with the following `rs.conf()` output:

```
{
    "_id" : "rs",
    "version" : 3,
    "members" : [
        {
            "_id" : 0,
            "host" : "database0.example.com:27017"
        },
        {
            "_id" : 1,
            "host" : "database1.example.com:27017"
        },
        {
            "_id" : 2,
            "host" : "database2.example.com:27017"
        }
    ]
}
```

The following procedures change the members' hostnames as follows:

- `mongodb0.example.net:27017` (the primary)
- `mongodb1.example.net:27017`
- `mongodb2.example.net:27017`

Use the most appropriate procedure for your deployment.

Change Hostnames while Maintaining Replica Set Availability

This procedure uses the above *assumptions* (page 203).

1. For each *secondary* in the replica set, perform the following sequence of operations:

   (a) Stop the secondary.

   (b) Restart the secondary at the new location.

   (c) Open a `mongo` shell connected to the replica set's primary. In our example, the primary runs on port `27017` so you would issue the following command:

   ```
   mongo --port 27017
   ```

   (d) Use `rs.reconfig()` to update the `replica set configuration document` with the new hostname.

   For example, the following sequence of commands updates the hostname for the secondary at the array index `1` of the `members` array (i.e. `members[1]`) in the replica set configuration document:

```
cfg = rs.conf()
cfg.members[1].host = "mongodb1.example.net:27017"
rs.reconfig(cfg)
```

For more information on updating the configuration document, see *replica-set-reconfiguration-usage*.

(e) Make sure your client applications are able to access the set at the new location and that the secondary has a chance to catch up with the other members of the set.

Repeat the above steps for each non-primary member of the set.

2. Open a `mongo` shell connected to the primary and step down the primary using the `rs.stepDown()` method:

```
rs.stepDown()
```

The replica set elects another member to the become primary.

3. When the step down succeeds, shut down the old primary.

4. Start the `mongod` instance that will become the new primary in the new location.

5. Connect to the current primary, which was just elected, and update the `replica set configuration document` with the hostname of the node that is to become the new primary.

For example, if the old primary was at position `0` and the new primary's hostname is `mongodb0.example.net:27017`, you would run:

```
cfg = rs.conf()
cfg.members[0].host = "mongodb0.example.net:27017"
rs.reconfig(cfg)
```

6. Open a `mongo` shell connected to the new primary.

7. To confirm the new configuration, call `rs.conf()` in the `mongo` shell.

Your output should resemble:

```
{
    "_id" : "rs",
    "version" : 4,
    "members" : [
        {
            "_id" : 0,
            "host" : "mongodb0.example.net:27017"
        },
        {
            "_id" : 1,
            "host" : "mongodb1.example.net:27017"
        },
        {
            "_id" : 2,
            "host" : "mongodb2.example.net:27017"
        }
    ]
}
```

Change All Hostnames at the Same Time

This procedure uses the above *assumptions* (page 203).

1. Stop all members in the *replica set*.

2. Restart each member *on a different port* and *without* using the `--replSet` run-time option. Changing the port number during maintenance prevents clients from connecting to this host while you perform maintenance. Use the member's usual `--dbpath`, which in this example is `/data/db1`. Use a command that resembles the following:

   ```
   mongod --dbpath /data/db1/ --port 37017
   ```

3. For each member of the replica set, perform the following sequence of operations:

   (a) Open a `mongo` shell connected to the `mongod` running on the new, temporary port. For example, for a member running on a temporary port of `37017`, you would issue this command:

   ```
   mongo --port 37017
   ```

   (b) Edit the replica set configuration manually. The replica set configuration is the only document in the `system.replset` collection in the `local` database. Edit the replica set configuration with the new hostnames and correct ports for all the members of the replica set. Consider the following sequence of commands to change the hostnames in a three-member set:

   ```
   use local

   cfg = db.system.replset.findOne( { "_id": "rs" } )

   cfg.members[0].host = "mongodb0.example.net:27017"

   cfg.members[1].host = "mongodb1.example.net:27017"

   cfg.members[2].host = "mongodb2.example.net:27017"

   db.system.replset.update( { "_id": "rs" } , cfg )
   ```

   (c) Stop the `mongod` process on the member.

4. After re-configuring all members of the set, start each `mongod` instance in the normal way: use the usual port number and use the `--replSet` option. For example:

   ```
   mongod --dbpath /data/db1/ --port 27017 --replSet rs
   ```

5. Connect to one of the `mongod` instances using the `mongo` shell. For example:

   ```
   mongo --port 27017
   ```

6. To confirm the new configuration, call `rs.conf()` in the `mongo` shell.

   Your output should resemble:

   ```
   {
       "_id" : "rs",
       "version" : 4,
       "members" : [
           {
               "_id" : 0,
               "host" : "mongodb0.example.net:27017"
           },
           {
               "_id" : 1,
               "host" : "mongodb1.example.net:27017"
           },
           {
               "_id" : 2,
               "host" : "mongodb2.example.net:27017"
   ```

```
                }
            ]
        }
```

Configure a Secondary's Sync Target

On this page

Overview

Secondaries capture data from the primary member to maintain an up to date copy of the sets' data. However, by default secondaries may automatically change their sync targets to secondary members based on changes in the ping time between members and the state of other members' replication. See `https://docs.mongodb.org/manual/core/replica-set-sync` and *Manage Chained Replication* (page 201) for more information.

For some deployments, implementing a custom replication sync topology may be more effective than the default sync target selection logic. MongoDB provides the ability to specify a host to use as a sync target.

To override the default sync target selection logic, you may manually configure a *secondary* member's sync target to temporarily pull *oplog* entries. The following provide access to this functionality:

- `replSetSyncFrom` command, or
- `rs.syncFrom()` helper in the `mongo` shell

Considerations

Sync Logic Only modify the default sync logic as needed, and always exercise caution. `rs.syncFrom()` will not affect an in-progress initial sync operation. To affect the sync target for the initial sync, run `rs.syncFrom()` operation *before* initial sync.

If you run `rs.syncFrom()` during initial sync, MongoDB produces no error messages, but the sync target will not change until after the initial sync operation.

Persistence `replSetSyncFrom` and `rs.syncFrom()` provide a temporary override of default behavior. `mongod` will revert to the default sync behavior in the following situations:

- The `mongod` instance restarts.
- The connection between the `mongod` and the sync target closes.

Changed in version 2.4: The sync target falls more than 30 seconds behind another member of the replica set; the `mongod` will revert to the default sync target.

Target The member to sync from must be a valid source for data in the set. To sync from a member, the member must:

- Have data. It cannot be an arbiter, in startup or recovering mode, and must be able to answer data queries.
- Be accessible.
- Be a member of the same set in the replica set configuration.
- Build indexes with the `buildIndexes` setting.
- A different member of the set, to prevent syncing from itself.

If you attempt to replicate from a member that is more than 10 seconds behind the current member, `mongod` will log a warning but will still replicate from the lagging member.

If you run `replSetSyncFrom` during initial sync, MongoDB produces no error messages, but the sync target will not change until after the initial sync operation.

Procedure

To use the `replSetSyncFrom` command in the `mongo` shell:

```
db.adminCommand( { replSetSyncFrom: "hostname<:port>" } );
```

To use the `rs.syncFrom()` helper in the `mongo` shell:

```
rs.syncFrom("hostname<:port>");
```

5.1.4 Troubleshoot Replica Sets

On this page

This section describes common strategies for troubleshooting *replica set* deployments.

Check Replica Set Status

To display the current state of the replica set and current state of each member, run the `rs.status()` method in a `mongo` shell connected to the replica set's *primary*. For descriptions of the information displayed by `rs.status()`, see `https://docs.mongodb.org/manual/reference/command/replSetGetStatus`.

Note: The `rs.status()` method is a wrapper that runs the `replSetGetStatus` database command.

Check the Replication Lag

Replication lag is a delay between an operation on the *primary* and the application of that operation from the *oplog* to the *secondary*. Replication lag can be a significant issue and can seriously affect MongoDB *replica set* deployments. Excessive replication lag makes "lagged" members ineligible to quickly become primary and increases the possibility that distributed read operations will be inconsistent.

To check the current length of replication lag:

- In a `mongo` shell connected to the primary, call the `rs.printSlaveReplicationInfo()` method.

  Returns the `syncedTo` value for each member, which shows the time when the last oplog entry was written to the secondary, as shown in the following example:

  ```
  source: m1.example.net:27017
      syncedTo: Thu Apr 10 2014 10:27:47 GMT-0400 (EDT)
      0 secs (0 hrs) behind the primary
  source: m2.example.net:27017
      syncedTo: Thu Apr 10 2014 10:27:47 GMT-0400 (EDT)
      0 secs (0 hrs) behind the primary
  ```

  A *delayed member* may show as `0` seconds behind the primary when the inactivity period on the primary is greater than the `slaveDelay` value.

  Note: The `rs.status()` method is a wrapper around the `replSetGetStatus` database command.

- Monitor the rate of replication by watching the oplog time in the "replica" graph in the MongoDB Cloud Manager[3] and in Ops Manager, an on-premise solution available in MongoDB Enterprise Advanced[4]. For more information see the MongoDB Cloud Manager documentation[5] and Ops Manager documentation[6].

Possible causes of replication lag include:

- **Network Latency**

  Check the network routes between the members of your set to ensure that there is no packet loss or network routing issue.

  Use tools including `ping` to test latency between set members and `traceroute` to expose the routing of packets network endpoints.

- **Disk Throughput**

  If the file system and disk device on the secondary is unable to flush data to disk as quickly as the primary, then the secondary will have difficulty keeping state. Disk-related issues are incredibly prevalent on multi-tenant systems, including virtualized instances, and can be transient if the system accesses disk devices over an IP network (as is the case with Amazon's EBS system.)

  Use system-level tools to assess disk status, including `iostat` or `vmstat`.

- **Concurrency**

  In some cases, long-running operations on the primary can block replication on secondaries. For best results, configure *write concern* to request confirmation of replication to secondaries. This prevents write operations from returning if replication cannot keep up with the write load.

  Use the *database profiler* to see if there are slow queries or long-running operations that correspond to the incidences of lag.

[3]https://cloud.mongodb.com/?jmp=docs
[4]https://www.mongodb.com/products/mongodb-enterprise-advanced?jmp=docs
[5]https://docs.cloud.mongodb.com/
[6]https://docs.opsmanager.mongodb.com/current/

- **Appropriate Write Concern**

  If you are performing a large data ingestion or bulk load operation that requires a large number of writes to the primary, particularly with `unacknowledged write concern`, the secondaries will not be able to read the oplog fast enough to keep up with changes.

  To prevent this, request `write acknowledgment write concern` after every 100, 1,000, or an another interval to provide an opportunity for secondaries to catch up with the primary.

  For more information see:

  – *Replica Acknowledge Write Concern*

  – *Replica Set Write Concern*

  – *replica-set-oplog-sizing*

Test Connections Between all Members

All members of a *replica set* must be able to connect to every other member of the set to support replication. Always verify connections in both "directions." Networking topologies and firewall configurations can prevent normal and required connectivity, which can block replication.

Consider the following example of a bidirectional test of networking:

Example

Given a replica set with three members running on three separate hosts:

- `m1.example.net`
- `m2.example.net`
- `m3.example.net`

1. Test the connection from `m1.example.net` to the other hosts with the following operation set `m1.example.net`:

   ```
   mongo --host m2.example.net --port 27017

   mongo --host m3.example.net --port 27017
   ```

2. Test the connection from `m2.example.net` to the other two hosts with the following operation set from `m2.example.net`, as in:

   ```
   mongo --host m1.example.net --port 27017

   mongo --host m3.example.net --port 27017
   ```

   You have now tested the connection between `m2.example.net` and `m1.example.net` in both directions.

3. Test the connection from `m3.example.net` to the other two hosts with the following operation set from the `m3.example.net` host, as in:

   ```
   mongo --host m1.example.net --port 27017

   mongo --host m2.example.net --port 27017
   ```

If any connection, in any direction fails, check your networking and firewall configuration and reconfigure your environment to allow these connections.

Socket Exceptions when Rebooting More than One Secondary

When you reboot members of a replica set, ensure that the set is able to elect a primary during the maintenance. This means ensuring that a majority of the set's `votes` are available.

When a set's active members can no longer form a majority, the set's *primary* steps down and becomes a *secondary*. The former primary closes all open connections to client applications. Clients attempting to write to the former primary receive socket exceptions and *Connection reset* errors until the set can elect a primary.

Example

Given a three-member replica set where every member has one vote, the set can elect a primary if at least two members can connect to each other. If you reboot the two secondaries at once, the primary steps down and becomes a secondary. Until at least another secondary becomes available, i.e. at least one of the rebooted secondaries also becomes available, the set has no primary and cannot elect a new primary.

For more information on votes, see `https://docs.mongodb.org/manual/core/replica-set-elections`. For related information on connection errors, see *faq-keepalive*.

Check the Size of the Oplog

A larger *oplog* can give a replica set a greater tolerance for lag, and make the set more resilient.

To check the size of the oplog for a given *replica set* member, connect to the member in a `mongo` shell and run the `rs.printReplicationInfo()` method.

The output displays the size of the oplog and the date ranges of the operations contained in the oplog. In the following example, the oplog is about 10MB and is able to fit about 26 hours (94400 seconds) of operations:

```
configured oplog size:   10.10546875MB
log length start to end: 94400 (26.22hrs)
oplog first event time:  Mon Mar 19 2012 13:50:38 GMT-0400 (EDT)
oplog last event time:   Wed Oct 03 2012 14:59:10 GMT-0400 (EDT)
now:                     Wed Oct 03 2012 15:00:21 GMT-0400 (EDT)
```

The oplog should be long enough to hold all transactions for the longest downtime you expect on a secondary. At a minimum, an oplog should be able to hold minimum 24 hours of operations; however, many users prefer to have 72 hours or even a week's work of operations.

For more information on how oplog size affects operations, see:

- *replica-set-oplog-sizing*,
- *replica-set-delayed-members*, and
- *Check the Replication Lag* (page 208).

Note: You normally want the oplog to be the same size on all members. If you resize the oplog, resize it on all members.

To change oplog size, see the *Change the Size of the Oplog* (page 187) tutorial.

Oplog Entry Timestamp Error

Consider the following error in `mongod` output and logs:

```
replSet error fatal couldn't query the local local.oplog.rs collection.  Terminating mongod after 30
<timestamp> [rsStart] bad replSet oplog entry?
```

Often, an incorrectly typed value in the `ts` field in the last *oplog* entry causes this error. The correct data type is Timestamp.

Check the type of the `ts` value using the following two queries against the oplog collection:

```
db = db.getSiblingDB("local")
db.oplog.rs.find().sort({$natural:-1}).limit(1)
db.oplog.rs.find({ts:{$type:17}}).sort({$natural:-1}).limit(1)
```

The first query returns the last document in the oplog, while the second returns the last document in the oplog where the `ts` value is a Timestamp. The `$type` operator allows you to select *BSON type* 17, is the Timestamp data type.

If the queries don't return the same document, then the last document in the oplog has the wrong data type in the `ts` field.

Example

If the first query returns this as the last oplog entry:

```
{ "ts" : {t: 1347982456000, i: 1},
  "h" : NumberLong("8191276672478122996"),
  "op" : "n",
  "ns" : "",
  "o" : { "msg" : "Reconfig set", "version" : 4 } }
```

And the second query returns this as the last entry where `ts` has the `Timestamp` type:

```
{ "ts" : Timestamp(1347982454000, 1),
  "h" : NumberLong("6188469075153256465"),
  "op" : "n",
  "ns" : "",
  "o" : { "msg" : "Reconfig set", "version" : 3 } }
```

Then the value for the `ts` field in the last oplog entry is of the wrong data type.

To set the proper type for this value and resolve this issue, use an update operation that resembles the following:

```
db.oplog.rs.update( { ts: { t:1347982456000, i:1 } },
                    { $set: { ts: new Timestamp(1347982456000, 1)}})
```

Modify the timestamp values as needed based on your oplog entry. This operation may take some period to complete because the update must scan and pull the entire oplog into memory.

Duplicate Key Error on `local.slaves`

Changed in version 3.0.0.

MongoDB 3.0.0 removes the `local.slaves` collection. For `local.slaves` error in earlier versions of MongoDB, refer to the appropriate version of the MongoDB Manual.

5.2 Sharded Cluster Tutorials

The following tutorials provide instructions for administering *sharded clusters*. For a higher-level overview, see `https://docs.mongodb.org/manual/sharding`.

***Sharded Cluster Deployment Tutorials* (page 212)** Instructions for deploying sharded clusters, adding shards, selecting shard keys, and the initial configuration of sharded clusters.

  ***Deploy a Sharded Cluster* (page 213)** Set up a sharded cluster by creating the needed data directories, starting the required MongoDB instances, and configuring the cluster settings.

  ***Considerations for Selecting Shard Keys* (page 216)** Choose the field that MongoDB uses to parse a collection's documents for distribution over the cluster's shards. Each shard holds documents with values within a certain range.

  ***Shard a Collection Using a Hashed Shard Key* (page 218)** Shard a collection based on hashes of a field's values in order to ensure even distribution over the collection's shards.

  ***Add Shards to a Cluster* (page 219)** Add a shard to add capacity to a sharded cluster.

  Continue reading from *Sharded Cluster Deployment Tutorials* (page 212) for additional tutorials.

***Sharded Cluster Maintenance Tutorials* (page 228)** Procedures and tasks for common operations on active sharded clusters.

  ***View Cluster Configuration* (page 228)** View status information about the cluster's databases, shards, and chunks.

  ***Remove Shards from an Existing Sharded Cluster* (page 243)** Migrate a single shard's data and remove the shard.

  ***Migrate Config Servers with Different Hostnames* (page 230)** Migrate a config server to a new system that uses a new hostname. If possible, avoid changing the hostname and instead use the *Migrate Config Servers with the Same Hostname* (page 230) procedure.

  ***Manage Shard Tags* (page 254)** Use tags to associate specific ranges of shard key values with specific shards.

  Continue reading from *Sharded Cluster Maintenance Tutorials* (page 228) for additional tutorials.

***Sharded Cluster Data Management* (page 245)** Practices that address common issues in managing large sharded data sets.

***Troubleshoot Sharded Clusters* (page 259)** Presents solutions to common issues and concerns relevant to the administration and use of sharded clusters. Refer to `https://docs.mongodb.org/manual/faq/diagnostics` for general diagnostic information.

5.2.1 Sharded Cluster Deployment Tutorials

The following tutorials provide information on deploying sharded clusters.

***Deploy a Sharded Cluster* (page 213)** Set up a sharded cluster by creating the needed data directories, starting the required MongoDB instances, and configuring the cluster settings.

***Considerations for Selecting Shard Keys* (page 216)** Choose the field that MongoDB uses to parse a collection's documents for distribution over the cluster's shards. Each shard holds documents with values within a certain range.

***Shard a Collection Using a Hashed Shard Key* (page 218)** Shard a collection based on hashes of a field's values in order to ensure even distribution over the collection's shards.

***Add Shards to a Cluster* (page 219)** Add a shard to add capacity to a sharded cluster.

***Deploy Three Config Servers for Production Deployments* (page 220)** Convert a test deployment with one config server to a production deployment with three config servers.

***Convert a Replica Set to a Replicated Sharded Cluster* (page 221)** Convert a replica set to a sharded cluster in which each shard is its own replica set.

***Convert Sharded Cluster to Replica Set* (page 227)** Replace your sharded cluster with a single replica set.

Deploy a Sharded Cluster

On this page

Use the following sequence of tasks to deploy a sharded cluster:

Warning: Sharding and "localhost" Addresses

If you use either "localhost" or `127.0.0.1` as the hostname portion of any host identifier, for example as the `host` argument to `addShard` or the value to the *`--configdb`* run time option, then you must use "localhost" or `127.0.0.1` for *all* host settings for any MongoDB instances in the cluster. If you mix localhost addresses and remote host address, MongoDB will error.

Start the Config Server Database Instances

The config server processes are `mongod` instances that store the cluster's metadata. You designate a `mongod` as a config server using the *`--configsvr`* option. Each config server stores a complete copy of the cluster's metadata.

In production deployments, you must deploy exactly three config server instances, each running on different servers to assure good uptime and data safety. In test environments, you can run all three instances on a single server.

Important: All members of a sharded cluster must be able to connect to *all* other members of a sharded cluster, including all shards and all config servers. Ensure that the network and security systems including all interfaces and firewalls, allow these connections.

1. Create data directories for each of the three config server instances. By default, a config server stores its data files in the */data/configdb* directory. You can choose a different location. To create a data directory, issue a command similar to the following:

   ```
   mkdir /data/configdb
   ```

2. Start the three config server instances. Start each by issuing a command using the following syntax:

   ```
   mongod --configsvr --dbpath <path> --port <port>
   ```

   The default port for config servers is `27019`. You can specify a different port. The following example starts a config server using the default port and default data directory:

   ```
   mongod --configsvr --dbpath /data/configdb --port 27019
   ```

   For additional command options, see `https://docs.mongodb.org/manual/reference/program/mongod` or `https://docs.mongodb.org/manual/reference/configuration-options`.

   Note: All config servers must be running and available when you first initiate a *sharded cluster*.

Start the `mongos` Instances

The `mongos` instances are lightweight and do not require data directories. You can run a `mongos` instance on a system that runs other cluster components, such as on an application server or a server running a `mongod` process. By default, a `mongos` instance runs on port `27017`.

When you start the `mongos` instance, specify the hostnames of the three config servers, either in the configuration file or as command line parameters.

Tip

To avoid downtime, give each config server a logical DNS name (unrelated to the server's physical or virtual hostname). Without logical DNS names, moving or renaming a config server requires shutting down every `mongod` and `mongos` instance in the sharded cluster.

To start a `mongos` instance, issue a command using the following syntax:

```
mongos --configdb <config server hostnames>
```

For example, to start a `mongos` that connects to config server instance running on the following hosts and on the default ports:

- `cfg0.example.net`
- `cfg1.example.net`
- `cfg2.example.net`

You would issue the following command:

```
mongos --configdb cfg0.example.net:27019,cfg1.example.net:27019,cfg2.example.net:27019
```

Each `mongos` in a sharded cluster must use the same `configDB` string, with identical host names listed in identical order.

If you start a `mongos` instance with a string that *does not* exactly match the string used by the other `mongos` instances in the cluster, the `mongos` instance returns a *Config Database String Error* (page 259) error and refuses to start.

Add Shards to the Cluster

A *shard* can be a standalone `mongod` or a *replica set*. In a production environment, each shard should be a replica set. Use the procedure in *Deploy a Replica Set* (page 160) to deploy replica sets for each shard.

1. From a `mongo` shell, connect to the `mongos` instance. Issue a command using the following syntax:

   ```
   mongo --host <hostname of machine running mongos> --port <port mongos listens on>
   ```

   For example, if a `mongos` is accessible at `mongos0.example.net` on port `27017`, issue the following command:

   ```
   mongo --host mongos0.example.net --port 27017
   ```

2. Add each shard to the cluster using the `sh.addShard()` method, as shown in the examples below. Issue `sh.addShard()` separately for each shard. If the shard is a replica set, specify the name of the replica set and specify a member of the set. In production deployments, all shards should be replica sets.

   Optional

You can instead use the `addShard` database command, which lets you specify a name and maximum size for the shard. If you do not specify these, MongoDB automatically assigns a name and maximum size. To use the database command, see `addShard`.

The following are examples of adding a shard with `sh.addShard()`:

- To add a shard for a replica set named `rs1` with a member running on port `27017` on `mongodb0.example.net`, issue the following command:

  ```
  sh.addShard( "rs1/mongodb0.example.net:27017" )
  ```

- To add a shard for a standalone `mongod` on port `27017` of `mongodb0.example.net`, issue the following command:

  ```
  sh.addShard( "mongodb0.example.net:27017" )
  ```

Note: It might take some time for *chunks* to migrate to the new shard.

Enable Sharding for a Database

Before you can shard a collection, you must enable sharding for the collection's database. Enabling sharding for a database does not redistribute data but make it possible to shard the collections in that database.

Once you enable sharding for a database, MongoDB assigns a *primary shard* for that database where MongoDB stores all data before sharding begins.

1. From a `mongo` shell, connect to the `mongos` instance. Issue a command using the following syntax:

   ```
   mongo --host <hostname of machine running mongos> --port <port mongos listens on>
   ```

2. Issue the `sh.enableSharding()` method, specifying the name of the database for which to enable sharding. Use the following syntax:

   ```
   sh.enableSharding("<database>")
   ```

Optionally, you can enable sharding for a database using the `enableSharding` command, which uses the following syntax:

```
db.runCommand( { enableSharding: <database> } )
```

Shard a Collection

You shard on a per-collection basis.

1. Determine what you will use for the *shard key*. Your selection of the shard key affects the efficiency of sharding. See the selection considerations listed in the *Considerations for Selecting Shard Key* (page 217).

2. If the collection already contains data you must create an index on the *shard key* using `createIndex()`. If the collection is empty then MongoDB will create the index as part of the `sh.shardCollection()` step.

3. Shard a collection by issuing the `sh.shardCollection()` method in the `mongo` shell. The method uses the following syntax:

   ```
   sh.shardCollection("<database>.<collection>", shard-key-pattern)
   ```

Replace the `<database>.<collection>` string with the full namespace of your database, which consists of the name of your database, a dot (e.g. `.`), and the full name of the collection. The `shard-key-pattern` represents your shard key, which you specify in the same form as you would an `index` key pattern.

Example

The following sequence of commands shards four collections:

```
sh.shardCollection("records.people", { "zipcode": 1, "name": 1 } )
sh.shardCollection("people.addresses", { "state": 1, "_id": 1 } )
sh.shardCollection("assets.chairs", { "type": 1, "_id": 1 } )
sh.shardCollection("events.alerts", { "_id": "hashed" } )
```

In order, these operations shard:

(a) The `people` collection in the `records` database using the shard key `{ "zipcode": 1, "name": 1 }`.

This shard key distributes documents by the value of the `zipcode` field. If a number of documents have the same value for this field, then that *chunk* will be *splittable* (page 217) by the values of the `name` field.

(b) The `addresses` collection in the `people` database using the shard key `{ "state": 1, "_id": 1 }`.

This shard key distributes documents by the value of the `state` field. If a number of documents have the same value for this field, then that *chunk* will be *splittable* (page 217) by the values of the `_id` field.

(c) The `chairs` collection in the `assets` database using the shard key `{ "type": 1, "_id": 1 }`.

This shard key distributes documents by the value of the `type` field. If a number of documents have the same value for this field, then that *chunk* will be *splittable* (page 217) by the values of the `_id` field.

(d) The `alerts` collection in the `events` database using the shard key `{ "_id": "hashed" }`.

New in version 2.4.

This shard key distributes documents by a hash of the value of the `_id` field. MongoDB computes the hash of the `_id` field for the *hashed index*, which should provide an even distribution of documents across a cluster.

Considerations for Selecting Shard Keys

Choosing a Shard Key

For many collections there may be no single, naturally occurring key that possesses all the qualities of a good shard key. The following strategies may help construct a useful shard key from existing data:

1. Compute a more ideal shard key in your application layer, and store this in all of your documents, potentially in the `_id` field.
2. Use a compound shard key that uses two or three values from all documents that provide the right mix of cardinality with scalable write operations and query isolation.
3. Determine that the impact of using a less than ideal shard key is insignificant in your use case, given:
   - limited write volume,
   - expected data size, or
   - application query patterns.

4. New in version 2.4: Use a *hashed shard key*. Choose a field that has high cardinality and create a *hashed index* on that field. MongoDB uses these hashed index values as shard key values, which ensures an even distribution of documents across the shards.

   Tip

   MongoDB automatically computes the hashes when resolving queries using hashed indexes. Applications do **not** need to compute hashes.

Considerations for Selecting Shard Key

Choosing the correct shard key can have a great impact on the performance, capability, and functioning of your database and cluster. Appropriate shard key choice depends on the schema of your data and the way that your applications query and write data.

Create a Shard Key that is Easily Divisible An easily divisible shard key makes it easy for MongoDB to distribute content among the shards. Shard keys that have a limited number of possible values can result in chunks that are "unsplittable".

For instance, if a chunk represents a single shard key value, then MongoDB cannot split the chunk even when the chunk exceeds the size at which `splits` occur.

See also:

Cardinality (page 217)

Create a Shard Key that has High Degree of Randomness A shard key with high degree of randomness prevents any single shard from becoming a bottleneck and will distribute write operations among the cluster.

See also:

sharding-shard-key-write-scaling

Create a Shard Key that Targets a Single Shard A shard key that targets a single shard makes it possible for the **mongos** program to return most query operations directly from a single *specific* **mongod** instance. Your shard key should be the primary field used by your queries. Fields with a high degree of "randomness" make it difficult to target operations to specific shards.

See also:

sharding-shard-key-query-isolation

Shard Using a Compound Shard Key The challenge when selecting a shard key is that there is not always an obvious choice. Often, an existing field in your collection may not be the optimal key. In those situations, computing a special purpose shard key into an additional field or using a compound shard key may help produce one that is more ideal.

Cardinality Cardinality in the context of MongoDB, refers to the ability of the system to *partition* data into *chunks*. For example, consider a collection of data such as an "address book" that stores address records:

- Consider the use of a `state` field as a shard key:

The state key's value holds the US state for a given address document. This field has a *low cardinality* as all documents that have the same value in the `state` field *must* reside on the same shard, even if a particular state's chunk exceeds the maximum chunk size.

Since there are a limited number of possible values for the `state` field, MongoDB may distribute data unevenly among a small number of fixed chunks. This may have a number of effects:

  - If MongoDB cannot split a chunk because all of its documents have the same shard key, migrations involving these un-splittable chunks will take longer than other migrations, and it will be more difficult for your data to stay balanced.
  - If you have a fixed maximum number of chunks, you will never be able to use more than that number of shards for this collection.

- Consider the use of a `zipcode` field as a shard key:

  While this field has a large number of possible values, and thus has potentially higher cardinality, it's possible that a large number of users could have the same value for the shard key, which would make this chunk of users un-splittable.

  In these cases, cardinality depends on the data. If your address book stores records for a geographically distributed contact list (e.g. "Dry cleaning businesses in America,") then a value like zipcode would be sufficient. However, if your address book is more geographically concentrated (e.g "ice cream stores in Boston Massachusetts,") then you may have a much lower cardinality.

- Consider the use of a `phone-number` field as a shard key:

  Phone number has a *high cardinality,* because users will generally have a unique value for this field, MongoDB will be able to split as many chunks as needed.

While "high cardinality," is necessary for ensuring an even distribution of data, having a high cardinality does not guarantee sufficient *query isolation* or appropriate *write scaling*.

If you choose a shard key with low cardinality, some chunks may grow too large for MongoDB to migrate. See *jumbo-chunks* for more information.

Shard Key Selection Strategy

When selecting a shard key, it is difficult to balance the qualities of an ideal shard key, which sometimes dictate opposing strategies. For instance, it's difficult to produce a key that has both a high degree randomness for even data distribution and a shard key that allows your application to target specific shards. For some workloads, it's more important to have an even data distribution, and for others targeted queries are essential.

Therefore, the selection of a shard key is about balancing both your data and the performance characteristics caused by different possible data distributions and system workloads.

Shard a Collection Using a Hashed Shard Key

On this page

New in version 2.4.

Hashed shard keys use a *hashed index* of a field as the *shard key* to partition data across your sharded cluster.

For suggestions on choosing the right field as your hashed shard key, see *sharding-hashed-sharding*. For limitations on hashed indexes, see *index-hashed-index*.

Note: If chunk migrations are in progress while creating a hashed shard key collection, the initial chunk distribution may be uneven until the balancer automatically balances the collection.

Shard the Collection

To shard a collection using a hashed shard key, use an operation in the `mongo` that resembles the following:

```
sh.shardCollection( "records.active", { a: "hashed" } )
```

This operation shards the `active` collection in the `records` database, using a hash of the `a` field as the shard key.

Specify the Initial Number of Chunks

If you shard an empty collection using a hashed shard key, MongoDB automatically creates and migrates empty chunks so that each shard has two chunks. To control how many chunks MongoDB creates when sharding the collection, use `shardCollection` with the `numInitialChunks` parameter.

Important: MongoDB 2.4 adds support for hashed shard keys. After sharding a collection with a hashed shard key, you must use the MongoDB 2.4 or higher `mongos` and `mongod` instances in your sharded cluster.

Warning: MongoDB `hashed` indexes truncate floating point numbers to 64-bit integers before hashing. For example, a `hashed` index would store the same value for a field that held a value of `2.3`, `2.2`, and `2.9`. To prevent collisions, do not use a `hashed` index for floating point numbers that cannot be reliably converted to 64-bit integers (and then back to floating point). MongoDB `hashed` indexes do not support floating point values larger than 2^{53}.

Add Shards to a Cluster

On this page

You add shards to a *sharded cluster* after you create the cluster or any time that you need to add capacity to the cluster. If you have not created a sharded cluster, see *Deploy a Sharded Cluster* (page 213).

In production environments, all shards should be *replica sets*.

Considerations

Balancing When you add a shard to a sharded cluster, you affect the balance of *chunks* among the shards of a cluster for all existing sharded collections. The balancer will begin migrating chunks so that the cluster will achieve balance. See `https://docs.mongodb.org/manual/core/sharding-balancing` for more information.

Changed in version 2.6: Chunk migrations can have an impact on disk space. Starting in MongoDB 2.6, the source shard automatically archives the migrated documents by default. For details, see *moveChunk-directory*.

Capacity Planning When adding a shard to a cluster, always ensure that the cluster has enough capacity to support the migration required for balancing the cluster without affecting legitimate production traffic.

Add a Shard to a Cluster

You interact with a sharded cluster by connecting to a `mongos` instance.

1. From a `mongo` shell, connect to the `mongos` instance. For example, if a `mongos` is accessible at `mongos0.example.net` on port `27017`, issue the following command:

   ```
   mongo --host mongos0.example.net --port 27017
   ```

2. Add a shard to the cluster using the `sh.addShard()` method, as shown in the examples below. Issue `sh.addShard()` separately for each shard. If the shard is a replica set, specify the name of the replica set and specify a member of the set. In production deployments, all shards should be replica sets.

   Optional

   You can instead use the `addShard` database command, which lets you specify a name and maximum size for the shard. If you do not specify these, MongoDB automatically assigns a name and maximum size. To use the database command, see `addShard`.

   The following are examples of adding a shard with `sh.addShard()`:

   - To add a shard for a replica set named `rs1` with a member running on port `27017` on `mongodb0.example.net`, issue the following command:

     ```
     sh.addShard( "rs1/mongodb0.example.net:27017" )
     ```

     Changed in version 2.0.3.

     For MongoDB versions prior to 2.0.3, you must specify all members of the replica set. For example:

     ```
     sh.addShard( "rs1/mongodb0.example.net:27017,mongodb1.example.net:27017,mongodb2.example.net
     ```

   - To add a shard for a standalone `mongod` on port `27017` of `mongodb0.example.net`, issue the following command:

     ```
       sh.addShard( "mongodb0.example.net:27017" )
     ```

   Note: It might take some time for *chunks* to migrate to the new shard.

Deploy Three Config Servers for Production Deployments

This procedure converts a test deployment with only one *config server* to a production deployment with three config servers.

Tip

Use CNAMEs to identify your config servers to the cluster so that you can rename and renumber your config servers without downtime.

For redundancy, all production `sharded clusters` should deploy three config servers on three different machines. Use a single config server only for testing deployments, never for production deployments. When you shift to production, upgrade immediately to three config servers.

To convert a test deployment with one config server to a production deployment with three config servers:

1. Shut down all existing MongoDB processes in the cluster. This includes:
   - all `mongod` instances or *replica sets* that provide your shards.
   - all `mongos` instances in your cluster.
2. Copy the entire `dbPath` file system tree from the existing config server to the two machines that will provide the additional config servers. These commands, issued on the system with the existing *config-database*, `mongo-config0.example.net` may resemble the following:

   ```
   rsync -az /data/configdb mongo-config1.example.net:/data/configdb
   rsync -az /data/configdb mongo-config2.example.net:/data/configdb
   ```

3. Start all three config servers, using the same invocation that you used for the single config server.

   ```
   mongod --configsvr
   ```

4. Restart all shard `mongod` and `mongos` processes.

Convert a Replica Set to a Replicated Sharded Cluster

On this page

Overview

This tutorial converts a single three-member replica set to a sharded cluster with two shards. Each shard is an independent three-member replica set. The procedure is as follows:

1. Create the initial three-member replica set and insert data into a collection. See *Set Up Initial Replica Set* (page 222).
2. Start the config databases and a `mongos`. See *Deploy Config Databases and mongos* (page 223).
3. Add the initial replica set as a shard. See *Add Initial Replica Set as a Shard* (page 223).
4. Create a second shard and add to the cluster. See *Add Second Shard* (page 223).
5. Shard the desired collection. See *Shard a Collection* (page 224).

Prerequisites

This tutorial uses a total of ten servers: one server for the `mongos` and three servers each for the first *replica set*, the second replica set, and the `config servers`.

Each server must have a resolvable domain, hostname, or IP address within your system.

The tutorial uses the default data directories (e.g. `/data/db` and `/data/configdb`). Create the appropriate directories with appropriate permissions. To use different paths, see `https://docs.mongodb.org/manual/reference/configuration-options`.

The tutorial uses the `default ports` (e.g. `27017` and `27019`). To use different ports, see `https://docs.mongodb.org/manual/reference/configuration-options`.

Considerations

In production deployments, use exactly **three** config servers. Each config server must be on a separate machine.

In development and testing environments, you can deploy a cluster with a single config server.

Procedures

Set Up Initial Replica Set This procedure creates the initial three-member replica set `rs0`. The replica set members are on the following hosts: `mongodb0.example.net`, `mongodb1.example.net`, and `mongodb2.example.net`.

Step 1: Start each member of the replica set with the appropriate options. For each member, start a `mongod`, specifying the replica set name through the `replSet` option. Include any other parameters specific to your deployment. For replication-specific parameters, see *cli-mongod-replica-set*.

```
mongod --replSet "rs0"
```

Repeat this step for the other two members of the `rs0` replica set.

Step 2: Connect a `mongo` shell to a replica set member. Connect a `mongo` shell to *one* member of the replica set (e.g. `mongodb0.example.net`)

```
mongo mongodb0.example.net
```

Step 3: Initiate the replica set. From the `mongo` shell, run `rs.initiate()` to initiate a replica set that consists of the current member.

```
rs.initiate()
```

Step 4: Add the remaining members to the replica set.

```
rs.add("mongodb1.example.net")
rs.add("mongodb2.example.net")
```

Step 5: Create and populate a new collection. The following step adds one million documents to the collection `test_collection` and can take several minutes depending on your system.

Issue the following operations on the primary of the replica set:

```
use test
var bulk = db.test_collection.initializeUnorderedBulkOp();
people = ["Marc", "Bill", "George", "Eliot", "Matt", "Trey", "Tracy", "Greg", "Steve", "Kristina", "
for(var i=0; i<1000000; i++){
   user_id = i;
   name = people[Math.floor(Math.random()*people.length)];
   number = Math.floor(Math.random()*10001);
   bulk.insert( { "user_id":user_id, "name":name, "number":number });
}
bulk.execute();
```

For more information on deploying a replica set, see *Deploy a Replica Set* (page 160).

Deploy Config Databases and mongos This procedure deploys the three config servers and the mongos. The config servers use the following hosts: mongodb7.example.net, mongodb8.example.net, and mongodb9.example.net; the mongos uses mongodb6.example.net.

Step 1: Start three config databases. On each mongodb7.example.net, mongodb8.example.net, and mongodb9.example.net server, start the config server using default data directory /data/configdb and the default port 27019:

```
mongod --configsvr
```

To modify the default settings or to include additional options specific to your deployment, see https://docs.mongodb.org/manual/reference/configuration-options.

Step 2: Start a mongos instance. On mongodb6.example.net, start the mongos specifying the config servers. The mongos runs on the default port 27017.

This tutorial specifies a small *--chunkSize* of 1 MB to test sharding with the test_collection created earlier.

Note: In production environments, do **not** use a small chunkSize size.

```
mongos --configdb mongodb07.example.net:27019,mongodb08.example.net:27019,mongodb09.example.net:2701
```

Add Initial Replica Set as a Shard The following procedure adds the initial replica set rs0 as a shard.

Step 1: Connect a mongo shell to the mongos.

```
mongo mongodb6.example.net:27017/admin
```

Step 2: Add the shard. Add a shard to the cluster with the sh.addShard method:

```
sh.addShard( "rs0/mongodb0.example.net:27017,mongodb1.example.net:27017,mongodb2.example.net:27017" )
```

Add Second Shard The following procedure deploys a new replica set rs1 for the second shard and adds it to the cluster. The replica set members are on the following hosts: mongodb3.example.net, mongodb4.example.net, and mongodb5.example.net.

Step 1: Start each member of the replica set with the appropriate options. For each member, start a mongod, specifying the replica set name through the replSet option. Include any other parameters specific to your deployment. For replication-specific parameters, see *cli-mongod-replica-set*.

```
mongod --replSet "rs1"
```

Repeat this step for the other two members of the rs1 replica set.

Step 2: Connect a mongo shell to a replica set member. Connect a mongo shell to *one* member of the replica set (e.g. mongodb3.example.net)

```
mongo mongodb3.example.net
```

Step 3: Initiate the replica set. From the `mongo` shell, run `rs.initiate()` to initiate a replica set that consists of the current member.

```
rs.initiate()
```

Step 4: Add the remaining members to the replica set. Add the remaining members with the `rs.add()` method.

```
rs.add("mongodb4.example.net")
rs.add("mongodb5.example.net")
```

Step 5: Connect a `mongo` shell to the `mongos`.

```
mongo mongodb6.example.net:27017/admin
```

Step 6: Add the shard. In a `mongo` shell connected to the `mongos`, add the shard to the cluster with the `sh.addShard()` method:

```
sh.addShard( "rs1/mongodb3.example.net:27017,mongodb4.example.net:27017,mongodb5.example.net:27017" )
```

Shard a Collection

Step 1: Connect a `mongo` shell to the `mongos`.

```
mongo mongodb6.example.net:27017/admin
```

Step 2: Enable sharding for a database. Before you can shard a collection, you must first enable sharding for the collection's database. Enabling sharding for a database does not redistribute data but makes it possible to shard the collections in that database.

The following operation enables sharding on the `test` database:

```
sh.enableSharding( "test" )
```

The operation returns the status of the operation:

```
{ "ok" : 1 }
```

Step 3: Determine the shard key. For the collection to shard, determine the shard key. The *shard key* determines how MongoDB distributes the documents between shards. Good shard keys:

- have values that are evenly distributed among all documents,
- group documents that are often accessed at the same time into contiguous chunks, and
- allow for effective distribution of activity among shards.

Once you shard a collection with the specified shard key, you **cannot** change the shard key. For more information on shard keys, see `https://docs.mongodb.org/manual/core/sharding-shard-key` and *Considerations for Selecting Shard Keys* (page 216).

This procedure will use the `number` field as the shard key for `test_collection`.

Step 4: Create an index on the shard key. Before sharding a non-empty collection, create an `index on the shard key`.

```
use test
db.test_collection.createIndex( { number : 1 } )
```

Step 5: Shard the collection. In the `test` database, shard the `test_collection`, specifying `number` as the shard key.

```
use test
sh.shardCollection( "test.test_collection", { "number" : 1 } )
```

The method returns the status of the operation:

```
{ "collectionsharded" : "test.test_collection", "ok" : 1 }
```

The `balancer` will redistribute chunks of documents when it next runs. As clients insert additional documents into this collection, the `mongos` will route the documents between the shards.

Step 6: Confirm the shard is balancing. To confirm balancing activity, run `db.stats()` or `db.printShardingStatus()` in the `test` database.

```
use test
db.stats()
db.printShardingStatus()
```

Example output of the `db.stats()`:

```
{
  "raw" : {
      "rs0/mongodb0.example.net:27017,mongodb1.example.net:27017,mongodb2.example.net:27017" : {
         "db" : "test",
         "collections" : 3,
         "objects" : 989316,
         "avgObjSize" : 111.99974123535857,
         "dataSize" : 110803136,
         "storageSize" : 174751744,
         "numExtents" : 14,
         "indexes" : 2,
         "indexSize" : 57370992,
         "fileSize" : 469762048,
         "nsSizeMB" : 16,
         "dataFileVersion" : {
            "major" : 4,
            "minor" : 5
         },
         "extentFreeList" : {
            "num" : 0,
            "totalSize" : 0
         },
         "ok" : 1
      },
      "rs1/mongodb3.example.net:27017,mongodb4.example.net:27017,mongodb5.example.net:27017" : {
         "db" : "test",
         "collections" : 3,
         "objects" : 14697,
         "avgObjSize" : 111.98258147921345,
         "dataSize" : 1645808,
         "storageSize" : 2809856,
         "numExtents" : 7,
         "indexes" : 2,
         "indexSize" : 1169168,
         "fileSize" : 67108864,
         "nsSizeMB" : 16,
         "dataFileVersion" : {
            "major" : 4,
            "minor" : 5
         },
         "extentFreeList" : {
            "num" : 0,
            "totalSize" : 0
         },
         "ok" : 1
      }
  },
  "objects" : 1004013,
  "avgObjSize" : 111,
  "dataSize" : 112448944,
  "storageSize" : 177561600,
  "numExtents" : 21,
  "indexes" : 4,
  "indexSize" : 58540160,
  "fileSize" : 536870912,
  "extentFreeList" : {
      "num" : 0,
      "totalSize" : 0
  },
  "ok" : 1
}
```

Example output of the `db.printShardingStatus()`:

```
--- Sharding Status ---
sharding version: {
   "_id" : 1,
   "version" : 4,
   "minCompatibleVersion" : 4,
   "currentVersion" : 5,
   "clusterId" : ObjectId("5446970c04ad5132c271597c")
}
shards:
   {  "_id" : "rs0",  "host" : "rs0/mongodb0.example.net:27017,mongodb1.example.net:27017,mongodb2.ex
   {  "_id" : "rs1",  "host" : "rs1/mongodb3.example.net:27017,mongodb4.example.net:27017,mongodb5.ex
databases:
   {  "_id" : "admin",  "partitioned" : false,  "primary" : "config" }
   {  "_id" : "test",  "partitioned" : true,  "primary" : "rs0" }

test.test_collection
      shard key: { "number" : 1 }
      chunks:
         rs1    5
         rs0    186
      too many chunks to print, use verbose if you want to force print
```

Run these commands for a second time to demonstrate that *chunks* are migrating from `rs0` to `rs1`.

Convert Sharded Cluster to Replica Set

On this page

This tutorial describes the process for converting a *sharded cluster* to a non-sharded *replica set*. To convert a replica set into a sharded cluster *Convert a Replica Set to a Replicated Sharded Cluster* (page 221). See the `https://docs.mongodb.org/manual/sharding` documentation for more information on sharded clusters.

Convert a Cluster with a Single Shard into a Replica Set

In the case of a *sharded cluster* with only one shard, that shard contains the full data set. Use the following procedure to convert that cluster into a non-sharded *replica set*:

1. Reconfigure the application to connect to the primary member of the replica set hosting the single shard that system will be the new replica set.
2. Optionally remove the `--shardsrv` option, if your `mongod` started with this option.

Tip

Changing the `--shardsrv` option will change the port that `mongod` listens for incoming connections on.

The single-shard cluster is now a non-sharded *replica set* that will accept read and write operations on the data set.

You may now decommission the remaining sharding infrastructure.

Convert a Sharded Cluster into a Replica Set

Use the following procedure to transition from a *sharded cluster* with more than one shard to an entirely new *replica set*.

1. With the *sharded cluster* running, *deploy a new replica set* (page 160) in addition to your sharded cluster. The replica set must have sufficient capacity to hold all of the data files from all of the current shards combined. Do not configure the application to connect to the new replica set until the data transfer is complete.
2. Stop all writes to the *sharded cluster*. You may reconfigure your application or stop all `mongos` instances. If you stop all `mongos` instances, the applications will not be able to read from the database. If you stop all `mongos` instances, start a temporary `mongos` instance on that applications cannot access for the data migration procedure.
3. Use *mongodump and mongorestore* (page 82) to migrate the data from the `mongos` instance to the new *replica set*.

   Note: Not all collections on all databases are necessarily sharded. Do not solely migrate the sharded collections. Ensure that all databases and all collections migrate correctly.

4. Reconfigure the application to use the non-sharded *replica set* instead of the `mongos` instance.

The application will now use the un-sharded *replica set* for reads and writes. You may now decommission the remaining unused sharded cluster infrastructure.

5.2.2 Sharded Cluster Maintenance Tutorials

The following tutorials provide information in maintaining sharded clusters.

***View Cluster Configuration* (page 228)** View status information about the cluster's databases, shards, and chunks.

***Migrate Config Servers with the Same Hostname* (page 230)** Migrate a config server to a new system while keeping the same hostname. This procedure requires changing the DNS entry to point to the new system.

***Migrate Config Servers with Different Hostnames* (page 230)** Migrate a config server to a new system that uses a new hostname. If possible, avoid changing the hostname and instead use the *Migrate Config Servers with the Same Hostname* (page 230) procedure.

***Replace Disabled Config Server* (page 231)** Replaces a config server that has become inoperable. This procedure assumes that the hostname does not change.

***Migrate a Sharded Cluster to Different Hardware* (page 232)** Migrate a sharded cluster to a different hardware system, for example, when moving a pre-production environment to production.

***Backup Cluster Metadata* (page 235)** Create a backup of a sharded cluster's metadata while keeping the cluster operational.

***Configure Behavior of Balancer Process in Sharded Clusters* (page 236)** Manage the balancer's behavior by scheduling a balancing window, changing size settings, or requiring replication before migration.

***Manage Sharded Cluster Balancer* (page 238)** View balancer status and manage balancer behavior.

***Remove Shards from an Existing Sharded Cluster* (page 243)** Migrate a single shard's data and remove the shard.

View Cluster Configuration

On this page

List Databases with Sharding Enabled

To list the databases that have sharding enabled, query the `databases` collection in the *config-database*. A database has sharding enabled if the value of the `partitioned` field is `true`. Connect to a `mongos` instance with a `mongo` shell, and run the following operation to get a full list of databases with sharding enabled:

```
use config
db.databases.find( { "partitioned": true } )
```

Example

You can use the following sequence of commands when to return a list of all databases in the cluster:

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

If this returns the following result set:

```
{ "_id" : "admin", "partitioned" : false, "primary" : "config" }
{ "_id" : "animals", "partitioned" : true, "primary" : "m0.example.net:30001" }
{ "_id" : "farms", "partitioned" : false, "primary" : "m1.example2.net:27017" }
```

Then sharding is only enabled for the `animals` database.

List Shards

To list the current set of configured shards, use the `listShards` command, as follows:

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

View Cluster Details

To view cluster details, issue `db.printShardingStatus()` or `sh.status()`. Both methods return the same output.

Example

In the following example output from `sh.status()`

- `sharding version` displays the version number of the shard metadata.
- `shards` displays a list of the `mongod` instances used as shards in the cluster.
- `databases` displays all databases in the cluster, including database that do not have sharding enabled.
- The `chunks` information for the `foo` database displays how many chunks are on each shard and displays the range of each chunk.

```
--- Sharding Status ---
  sharding version: { "_id" : 1, "version" : 3 }
  shards:
    {  "_id" : "shard0000",  "host" : "m0.example.net:30001" }
    {  "_id" : "shard0001",  "host" : "m3.example2.net:50000" }
  databases:
    {  "_id" : "admin",  "partitioned" : false,  "primary" : "config" }
    {  "_id" : "contacts",  "partitioned" : true,  "primary" : "shard0000" }
        foo.contacts
            shard key: { "zip" : 1 }
            chunks:
                shard0001    2
                shard0002    3
                shard0000    2
            { "zip" : { "$minKey" : 1 } } -->> { "zip" : "56000" } on : shard0001 { "t" : 2, "i" : 0
            { "zip" : 56000 } -->> { "zip" : "56800" } on : shard0002 { "t" : 3, "i" : 4 }
            { "zip" : 56800 } -->> { "zip" : "57088" } on : shard0002 { "t" : 4, "i" : 2 }
            { "zip" : 57088 } -->> { "zip" : "57500" } on : shard0002 { "t" : 4, "i" : 3 }
            { "zip" : 57500 } -->> { "zip" : "58140" } on : shard0001 { "t" : 4, "i" : 0 }
            { "zip" : 58140 } -->> { "zip" : "59000" } on : shard0000 { "t" : 4, "i" : 1 }
            { "zip" : 59000 } -->> { "zip" : { "$maxKey" : 1 } } on : shard0000 { "t" : 3, "i" : 3 }
    {  "_id" : "test",  "partitioned" : false,  "primary" : "shard0000" }
```

Migrate Config Servers with the Same Hostname

This procedure migrates a *config server* in a `sharded cluster` to a new system that uses *the same* hostname.

To migrate all the config servers in a cluster, perform this procedure for each config server separately and migrate the config servers in reverse order from how they are listed in the `mongos` instances' `configDB` string. Start with the last config server listed in the `configDB` string.

1. Shut down the config server.

   This renders all config data for the sharded cluster "read only."

2. Change the DNS entry that points to the system that provided the old config server, so that the *same* hostname points to the new system. How you do this depends on how you organize your DNS and hostname resolution services.

3. Copy the contents of `dbPath` from the old config server to the new config server.

   For example, to copy the contents of `dbPath` to a machine named `mongodb.config2.example.net`, you might issue a command similar to the following:

   ```
   rsync -az /data/configdb/ mongodb.config2.example.net:/data/configdb
   ```

4. Start the config server instance on the new system. The default invocation is:

   ```
   mongod --configsvr
   ```

When you start the third config server, your cluster will become writable and it will be able to create new splits and migrate chunks as needed.

Migrate Config Servers with Different Hostnames

On this page

Overview

Sharded clusters use a group of three config servers to store cluster meta data, and all three config servers must be available to support cluster metadata changes that include chunk splits and migrations. If one of the config servers is unavailable or inoperable, you must replace it as soon as possible.

This procedure migrates a *config server* in a `sharded cluster` to a new server that uses a different hostname. Use this procedure only if the config server *will not* be accessible via the same hostname. If possible, avoid changing the hostname so that you can instead use the procedure to *migrate a config server and use the same hostname* (page 230).

Considerations

Changing a *config server's* hostname **requires downtime** and requires restarting every process in the sharded cluster.

While migrating config servers, always make sure that all `mongos` instances have three config servers specified in the `configDB` setting at all times. Also ensure that you specify the config servers in the same order for each `mongos` instance's `configDB` setting.

Procedure

1. Disable the cluster balancer process temporarily. See *Disable the Balancer* (page 240) for more information.
2. Shut down the config server to migrate.

   This renders all config data for the sharded cluster "read only."
3. Copy the contents of `dbPath` from the old config server to the new config server. For example, to copy the contents of `dbPath` to a machine named `mongodb.config2.example.net`, use a command that resembles the following:

   ```
   rsync -az /data/configdb mongodb.config2.example.net:/data/configdb
   ```
4. Start the config server instance on the new system. The default invocation is:

   ```
   mongod --configsvr
   ```
5. Shut down all existing MongoDB processes. This includes:
   - the `mongod` instances for the shards.
   - the `mongod` instances for the existing *config databases*.
   - the `mongos` instances.
6. Restart all shard `mongod` instances.
7. Restart the `mongod` instances for the two existing non-migrated config servers.
8. Update the `configDB` setting for each `mongos` instances.
9. Restart the `mongos` instances.
10. Re-enable the balancer to allow the cluster to resume normal balancing operations. See the *Disable the Balancer* (page 240) section for more information on managing the balancer process.

Replace Disabled Config Server

On this page

Overview

Sharded clusters use a group of three config servers to store cluster meta data, and all three config servers must be available to support cluster metadata changes that include chunk splits and migrations. If one of the config servers is unavailable or inoperable you must replace it as soon as possible.

This procedure replaces an inoperable *config server* in a `sharded cluster`. Use this procedure only to replace a config server that has become inoperable (e.g. hardware failure).

This process assumes that the hostname of the instance will not change. If you must change the hostname of the instance, use the procedure to *migrate a config server and use a new hostname* (page 230).

Considerations

In the course of this procedure *never* remove a config server from the `configDB` parameter on any of the `mongos` instances.

Procedure

Step 1: Provision a new system, with the same IP address and hostname as the previous host. You will have to ensure the new system has the same IP address and hostname as the system it's replacing *or* you will need to modify the DNS records and wait for them to propagate.

Step 2: Shut down *one* of the remaining config servers. Copy all of this host's `dbPath` path from the current system to the system that will provide the new config server. This command, issued on the system with the data files, may resemble the following:

```
rsync -az /data/configdb mongodb.config2.example.net:/data/configdb
```

Step 3: If necessary, update DNS and/or networking. Ensure the new config server is accessible by the same name as the previous config server.

Step 4: Start the *new* config server.

```
mongod --configsvr
```

Migrate a Sharded Cluster to Different Hardware

On this page

This procedure moves the components of the *sharded cluster* to a new hardware system without downtime for reads and writes.

Important: While the migration is in progress, do not attempt to change to the `cluster metadata`. Do not use any operation that modifies the cluster metadata *in any way*. For example, do not create or drop databases, create or drop collections, or use any sharding commands.

If your cluster includes a shard backed by a *standalone* `mongod` instance, consider *converting the standalone to a replica set* (page 172) to simplify migration and to let you keep the cluster online during future maintenance. Migrating a shard as standalone is a multi-step process that may require downtime.

To migrate a cluster to new hardware, perform the following tasks.

Disable the Balancer

Disable the balancer to stop `chunk migration` and do not perform any metadata write operations until the process finishes. If a migration is in progress, the balancer will complete the in-progress migration before stopping.

To disable the balancer, connect to one of the cluster's `mongos` instances and issue the following method:

```
sh.stopBalancer()
```

To check the balancer state, issue the `sh.getBalancerState()` method.

For more information, see *Disable the Balancer* (page 240).

Migrate Each Config Server Separately

Migrate each *config server* by starting with the *last* config server listed in the `configDB` string. Proceed in reverse order of the `configDB` string. Migrate and restart a config server before proceeding to the next. Do not rename a config server during this process.

Note: If the name or address that a sharded cluster uses to connect to a config server changes, you must restart **every** `mongod` and `mongos` instance in the sharded cluster. Avoid downtime by using CNAMEs to identify config servers within the MongoDB deployment.

See *Migrate Config Servers with Different Hostnames* (page 230) for more information.

Important: Start with the *last* config server listed in `configDB`.

1. Shut down the config server.

   This renders all config data for the sharded cluster "read only."

2. Change the DNS entry that points to the system that provided the old config server, so that the *same* hostname points to the new system. How you do this depends on how you organize your DNS and hostname resolution services.

3. Copy the contents of `dbPath` from the old config server to the new config server.

   For example, to copy the contents of `dbPath` to a machine named `mongodb.config2.example.net`, you might issue a command similar to the following:

   ```
   rsync -az /data/configdb/ mongodb.config2.example.net:/data/configdb
   ```

4. Start the config server instance on the new system. The default invocation is:

   ```
   mongod --configsvr
   ```

Restart the `mongos` Instances

If the `configDB` string will change as part of the migration, you must shut down *all* `mongos` instances before changing the `configDB` string. This avoids errors in the sharded cluster over `configDB` string conflicts.

If the `configDB` string will remain the same, you can migrate the `mongos` instances sequentially or all at once.

1. Shut down the `mongos` instances using the `shutdown` command. If the `configDB` string is changing, shut down *all* `mongos` instances.

2. If the hostname has changed for any of the config servers, update the `configDB` string for each `mongos` instance. The `mongos` instances must all use the same `configDB` string. The strings must list identical host names in identical order.

   Tip

   To avoid downtime, give each config server a logical DNS name (unrelated to the server's physical or virtual hostname). Without logical DNS names, moving or renaming a config server requires shutting down every `mongod` and `mongos` instance in the sharded cluster.

3. Restart the `mongos` instances being sure to use the updated `configDB` string if hostnames have changed.

For more information, see *Start the mongos Instances* (page 214).

Migrate the Shards

Migrate the shards one at a time. For each shard, follow the appropriate procedure in this section.

Migrate a Replica Set Shard To migrate a sharded cluster, migrate each member separately. First migrate the non-primary members, and then migrate the *primary* last.

If the replica set has two voting members, add an `arbiter` to the replica set to ensure the set keeps a majority of its votes available during the migration. You can remove the arbiter after completing the migration.

Migrate a Member of a Replica Set Shard

1. Shut down the `mongod` process. To ensure a clean shutdown, use the `shutdown` command.
2. Move the data directory (i.e., the `dbPath`) to the new machine.
3. Restart the `mongod` process at the new location.
4. Connect to the replica set's current primary.
5. If the hostname of the member has changed, use `rs.reconfig()` to update the `replica set configuration document` with the new hostname.

   For example, the following sequence of commands updates the hostname for the instance at position `2` in the `members` array:

   ```
   cfg = rs.conf()
   cfg.members[2].host = "pocatello.example.net:27017"
   rs.reconfig(cfg)
   ```

   For more information on updating the configuration document, see *replica-set-reconfiguration-usage*.
6. To confirm the new configuration, issue `rs.conf()`.
7. Wait for the member to recover. To check the member's state, issue `rs.status()`.

Migrate the Primary in a Replica Set Shard While migrating the replica set's primary, the set must elect a new primary. This failover process which renders the replica set unavailable to perform reads or accept writes for the duration of the election, which typically completes quickly. If possible, plan the migration during a maintenance window.

1. Step down the primary to allow the normal *failover* process. To step down the primary, connect to the primary and issue the either the `replSetStepDown` command or the `rs.stepDown()` method. The following example shows the `rs.stepDown()` method:

```
rs.stepDown()
```

2. Once the primary has stepped down and another member has become `PRIMARY` state. To migrate the stepped-down primary, follow the *Migrate a Member of a Replica Set Shard* (page 234) procedure

   You can check the output of `rs.status()` to confirm the change in status.

Migrate a Standalone Shard The ideal procedure for migrating a standalone shard is to *convert the standalone to a replica set* (page 172) and then use the procedure for *migrating a replica set shard* (page 234). In production clusters, all shards should be replica sets, which provides continued availability during maintenance windows.

Migrating a shard as standalone is a multi-step process during which part of the shard may be unavailable. If the shard is the *primary shard* for a database,the process includes the `movePrimary` command. While the `movePrimary` runs, you should stop modifying data in that database. To migrate the standalone shard, use the *Remove Shards from an Existing Sharded Cluster* (page 243) procedure.

Re-Enable the Balancer

To complete the migration, re-enable the balancer to resume `chunk migrations`.

Connect to one of the cluster's `mongos` instances and pass `true` to the `sh.setBalancerState()` method:

```
sh.setBalancerState(true)
```

To check the balancer state, issue the `sh.getBalancerState()` method.

For more information, see *Enable the Balancer* (page 241).

Backup Cluster Metadata

This procedure shuts down the `mongod` instance of a *config server* in order to create a backup of a `sharded cluster's` metadata. The cluster's config servers store all of the cluster's metadata, most importantly the mapping from *chunks* to *shards*.

When you perform this procedure, the cluster remains operational [7].

1. Disable the cluster balancer process temporarily. See *Disable the Balancer* (page 240) for more information.
2. Shut down one of the config databases.
3. Create a full copy of the data files (i.e. the path specified by the `dbPath` option for the config instance.)
4. Restart the original configuration server.
5. Re-enable the balancer to allow the cluster to resume normal balancing operations. See the *Disable the Balancer* (page 240) section for more information on managing the balancer process.

See also:

MongoDB Backup Methods (page 4).

[7] While one of the three config servers is unavailable, the cluster cannot split any chunks nor can it migrate chunks between shards. Your application will be able to write data to the cluster. See *sharding-config-server* for more information.

Configure Behavior of Balancer Process in Sharded Clusters

On this page

The balancer is a process that runs on *one* of the `mongos` instances in a cluster and ensures that *chunks* are evenly distributed throughout a sharded cluster. In most deployments, the default balancer configuration is sufficient for normal operation. However, administrators might need to modify balancer behavior depending on application or operational requirements. If you encounter a situation where you need to modify the behavior of the balancer, use the procedures described in this document.

For conceptual information about the balancer, see *sharding-balancing* and *sharding-balancing-internals*.

Schedule a Window of Time for Balancing to Occur

You can schedule a window of time during which the balancer can migrate chunks, as described in the following procedures:

- *Schedule the Balancing Window* (page 239)
- *Remove a Balancing Window Schedule* (page 240).

The `mongos` instances use their own local timezones when respecting balancer window.

Configure Default Chunk Size

The default chunk size for a sharded cluster is 64 megabytes. In most situations, the default size is appropriate for splitting and migrating chunks. For information on how chunk size affects deployments, see details, see *sharding-chunk-size*.

Changing the default chunk size affects chunks that are processes during migrations and auto-splits but does not retroactively affect all chunks.

To configure default chunk size, see *Modify Chunk Size in a Sharded Cluster* (page 251).

Change the Maximum Storage Size for a Given Shard

The `maxSize` field in the `shards` collection in the *config database* sets the maximum size for a shard, allowing you to control whether the balancer will migrate chunks to a shard. If `mapped` size [8] is above a shard's `maxSize`, the balancer will not move chunks to the shard. Also, the balancer will not move chunks off an overloaded shard. This must happen manually. The `maxSize` value only affects the balancer's selection of destination shards.

By default, `maxSize` is not specified, allowing shards to consume the total amount of available space on their machines if necessary.

You can set `maxSize` both when adding a shard and once a shard is running.

[8] This value includes the mapped size of all data files including the``local`` and `admin` databases. Account for this when setting `maxSize`.

To set `maxSize` when adding a shard, set the `addShard` command's `maxSize` parameter to the maximum size in megabytes. For example, the following command run in the `mongo` shell adds a shard with a maximum size of 125 megabytes:

```
db.runCommand( { addshard : "example.net:34008", maxSize : 125 } )
```

To set `maxSize` on an existing shard, insert or update the `maxSize` field in the `shards` collection in the *config database*. Set the `maxSize` in megabytes.

Example

Assume you have the following shard without a `maxSize` field:

```
{ "_id" : "shard0000", "host" : "example.net:34001" }
```

Run the following sequence of commands in the `mongo` shell to insert a `maxSize` of 125 megabytes:

```
use config
db.shards.update( { _id : "shard0000" }, { $set : { maxSize : 125 } } )
```

To later increase the `maxSize` setting to 250 megabytes, run the following:

```
use config
db.shards.update( { _id : "shard0000" }, { $set : { maxSize : 250 } } )
```

Change Replication Behavior for Chunk Migration

Secondary Throttle Changed in version 3.0.0: The balancer configuration document added configurable `writeConcern` to control the semantics of the `_secondaryThrottle` option.

The `_secondaryThrottle` parameter of the balancer and the `moveChunk` command affects the replication behavior during *chunk migration*. By default, `_secondaryThrottle` is `true`, which means each document move during chunk migration propagates to at least one secondary before the balancer proceeds with the next document: this is equivalent to a write concern of `{ w: 2 }`.

You can also configure the `writeConcern` for the `_secondaryThrottle` operation, to configure how migrations will wait for replication to complete. For more information on the replication behavior during various steps of chunk migration, see *chunk-migration-replication*.

To change the balancer's `_secondaryThrottle` and `writeConcern` values, connect to a `mongos` instance and directly update the `_secondaryThrottle` value in the `settings` collection of the *config database*. For example, from a `mongo` shell connected to a `mongos`, issue the following command:

```
use config
db.settings.update(
   { "_id" : "balancer" },
   { $set : { "_secondaryThrottle" : false ,
              "writeConcern": { "w": "majority" } } },
   { upsert : true }
)
```

The effects of changing the `_secondaryThrottle` and `writeConcern` value may not be immediate. To ensure an immediate effect, stop and restart the balancer to enable the selected value of `_secondaryThrottle`. See *Manage Sharded Cluster Balancer* (page 238) for details.

Wait for Delete The `_waitForDelete` setting of the balancer and the `moveChunk` command affects how the balancer migrates multiple chunks from a shard. By default, the balancer does not wait for the on-going migration's delete phase to complete before starting the next chunk migration. To have the delete phase **block** the start of the next chunk migration, you can set the `_waitForDelete` to true.

For details on chunk migration, see *sharding-chunk-migration*. For details on the chunk migration queuing behavior, see *chunk-migration-queuing*.

The `_waitForDelete` is generally for internal testing purposes. To change the balancer's `_waitForDelete` value:

1. Connect to a `mongos` instance.
2. Update the `_waitForDelete` value in the `settings` collection of the *config database*. For example:

```
use config
db.settings.update(
   { "_id" : "balancer" },
   { $set : { "_waitForDelete" : true } },
   { upsert : true }
)
```

Once set to `true`, to revert to the default behavior:

1. Connect to a `mongos` instance.
2. Update or unset the `_waitForDelete` field in the `settings` collection of the *config database*:

```
use config
db.settings.update(
   { "_id" : "balancer", "_waitForDelete": true },
   { $unset : { "_waitForDelete" : "" } }
)
```

Manage Sharded Cluster Balancer

On this page

This page describes common administrative procedures related to balancing. For an introduction to balancing, see *sharding-balancing*. For lower level information on balancing, see *sharding-balancing-internals*.

See also:

Configure Behavior of Balancer Process in Sharded Clusters (page 236)

Check the Balancer State

The following command checks if the balancer is enabled (i.e. that the balancer is allowed to run). The command does not check if the balancer is active (i.e. if it is actively balancing chunks).

To see if the balancer is enabled in your *cluster*, issue the following command, which returns a boolean:

```
sh.getBalancerState()
```

New in version 3.0.0: You can also see if the balancer is enabled using `sh.status()`. The `currently-enabled` field indicates whether the balancer is enabled, while the `currently-running` field indicates if the balancer is currently running.

Check the Balancer Lock

To see if the balancer process is active in your *cluster*, do the following:

1. Connect to any `mongos` in the cluster using the `mongo` shell.
2. Issue the following command to switch to the *config-database*:

   ```
   use config
   ```

3. Use the following query to return the balancer lock:

   ```
   db.locks.find( { _id : "balancer" } ).pretty()
   ```

When this command returns, you will see output like the following:

```
{   "_id" : "balancer",
"process" : "mongos0.example.net:1292810611:1804289383",
  "state" : 2,
     "ts" : ObjectId("4d0f872630c42d1978be8a2e"),
   "when" : "Mon Dec 20 2010 11:41:10 GMT-0500 (EST)",
    "who" : "mongos0.example.net:1292810611:1804289383:Balancer:846930886",
    "why" : "doing balance round" }
```

This output confirms that:

- The balancer originates from the `mongos` running on the system with the hostname `mongos0.example.net`.
- The value in the `state` field indicates that a `mongos` has the lock. For version 2.0 and later, the value of an active lock is `2`; for earlier versions the value is `1`.

Schedule the Balancing Window

In some situations, particularly when your data set grows slowly and a migration can impact performance, it's useful to be able to ensure that the balancer is active only at certain times. Use the following procedure to specify a window during which the *balancer* will be able to migrate chunks:

1. Connect to any `mongos` in the cluster using the `mongo` shell.
2. Issue the following command to switch to the *config-database*:

   ```
   use config
   ```

3. Issue the following operation to ensure the balancer is not in the `stopped` state:

```
sh.setBalancerState( true )
```

The balancer will not activate if in the `stopped` state or outside the `activeWindow` timeframe.

4. Use an operation modeled on the following `update()` operation to modify the balancer's window:

```
db.settings.update(
   { _id: "balancer" },
   { $set: { activeWindow : { start : "<start-time>", stop : "<stop-time>" } } },
   { upsert: true }
)
```

Replace `<start-time>` and `<end-time>` with time values using two digit hour and minute values (i.e. `HH:MM`) that specify the beginning and end boundaries of the balancing window.

- For `HH` values, use hour values ranging from `00` - `23`.
- For `MM` value, use minute values ranging from `00` - `59`.

The start and stop times will be evaluated relative to the time zone of each individual `mongos` instance in the sharded cluster. If your `mongos` instances are physically located in different time zones, use a common time zone (e.g. GMT) to ensure that the balancer window is interpreted correctly.

For instance, running the following will force the balancer to run between 11PM and 6AM local time only:

```
db.settings.update(
   { _id: "balancer" },
   { $set: { activeWindow : { start: "23:00", stop: "6:00" } } },
   { upsert: true }
)
```

Note: The balancer window must be sufficient to *complete* the migration of all data inserted during the day.

As data insert rates can change based on activity and usage patterns, it is important to ensure that the balancing window you select will be sufficient to support the needs of your deployment.

Do not use the `sh.startBalancer()` method when you have set an `activeWindow`.

Remove a Balancing Window Schedule

If you have *set the balancing window* (page 239) and wish to remove the schedule so that the balancer is always running, issue the following sequence of operations:

```
use config
db.settings.update({ _id : "balancer" }, { $unset : { activeWindow : true } })
```

Disable the Balancer

By default the balancer may run at any time and only moves chunks as needed. To disable the balancer for a short period of time and prevent all migration, use the following procedure:

1. Connect to any `mongos` in the cluster using the `mongo` shell.
2. Issue the following operation to disable the balancer:

```
sh.stopBalancer()
```

If a migration is in progress, the system will complete the in-progress migration before stopping.

3. To verify that the balancer will not start, issue the following command, which returns `false` if the balancer is disabled:

   ```
   sh.getBalancerState()
   ```

   Optionally, to verify no migrations are in progress after disabling, issue the following operation in the `mongo` shell:

   ```
   use config
   while( sh.isBalancerRunning() ) {
             print("waiting...");
             sleep(1000);
   }
   ```

Note: To disable the balancer from a driver that does not have the `sh.stopBalancer()` or `sh.setBalancerState()` helpers, issue the following command from the `config` database:

```
db.settings.update( { _id: "balancer" }, { $set : { stopped: true } } ,  { upsert: true } )
```

Enable the Balancer

Use this procedure if you have disabled the balancer and are ready to re-enable it:

1. Connect to any `mongos` in the cluster using the `mongo` shell.
2. Issue one of the following operations to enable the balancer:

   From the `mongo` shell, issue:

   ```
   sh.setBalancerState(true)
   ```

   From a driver that does not have the `sh.startBalancer()` helper, issue the following from the `config` database:

   ```
   db.settings.update( { _id: "balancer" }, { $set : { stopped: false } } , { upsert: true } )
   ```

Disable Balancing During Backups

If MongoDB migrates a *chunk* during a *backup* (page 4), you can end with an inconsistent snapshot of your *sharded cluster*. Never run a backup while the balancer is active. To ensure that the balancer is inactive during your backup operation:

- Set the *balancing window* (page 239) so that the balancer is inactive during the backup. Ensure that the backup can complete while you have the balancer disabled.
- *manually disable the balancer* (page 240) for the duration of the backup procedure.

If you turn the balancer off while it is in the middle of a balancing round, the shut down is not instantaneous. The balancer completes the chunk move in-progress and then ceases all further balancing rounds.

Before starting a backup operation, confirm that the balancer is not active. You can use the following command to determine if the balancer is active:

```
!sh.getBalancerState() && !sh.isBalancerRunning()
```

When the backup procedure is complete you can reactivate the balancer process.

Disable Balancing on a Collection

You can disable balancing for a specific collection with the `sh.disableBalancing()` method. You may want to disable the balancer for a specific collection to support maintenance operations or atypical workloads, for example, during data ingestions or data exports.

When you disable balancing on a collection, MongoDB will not interrupt in progress migrations.

To disable balancing on a collection, connect to a `mongos` with the `mongo` shell and call the `sh.disableBalancing()` method.

For example:

```
sh.disableBalancing("students.grades")
```

The `sh.disableBalancing()` method accepts as its parameter the full *namespace* of the collection.

Enable Balancing on a Collection

You can enable balancing for a specific collection with the `sh.enableBalancing()` method.

When you enable balancing for a collection, MongoDB will not *immediately* begin balancing data. However, if the data in your sharded collection is not balanced, MongoDB will be able to begin distributing the data more evenly.

To enable balancing on a collection, connect to a `mongos` with the `mongo` shell and call the `sh.enableBalancing()` method.

For example:

```
sh.enableBalancing("students.grades")
```

The `sh.enableBalancing()` method accepts as its parameter the full *namespace* of the collection.

Confirm Balancing is Enabled or Disabled

To confirm whether balancing for a collection is enabled or disabled, query the `collections` collection in the `config` database for the collection *namespace* and check the `noBalance` field. For example:

```
db.getSiblingDB("config").collections.findOne({_id : "students.grades"}).noBalance;
```

This operation will return a null error, `true`, `false`, or no output:

- A null error indicates the collection namespace is incorrect.
- If the result is `true`, balancing is disabled.
- If the result is `false`, balancing is enabled currently but has been disabled in the past for the collection. Balancing of this collection will begin the next time the balancer runs.
- If the operation returns no output, balancing is enabled currently and has never been disabled in the past for this collection. Balancing of this collection will begin the next time the balancer runs.

New in version 3.0.0: You can also see if the balancer is enabled using `sh.status()`. The `currently-enabled` field indicates if the balancer is enabled.

Remove Shards from an Existing Sharded Cluster

On this page

To remove a *shard* you must ensure the shard's data is migrated to the remaining shards in the cluster. This procedure describes how to safely migrate data and how to remove a shard.

This procedure describes how to safely remove a *single* shard. *Do not* use this procedure to migrate an entire cluster to new hardware. To migrate an entire shard to new hardware, migrate individual shards as if they were independent replica sets.

To remove a shard, first connect to one of the cluster's mongos instances using mongo shell. Then use the sequence of tasks in this document to remove a shard from the cluster.

Ensure the Balancer Process is Enabled

To successfully migrate data from a shard, the *balancer* process **must** be enabled. Check the balancer state using the sh.getBalancerState() helper in the mongo shell. For more information, see the section on *balancer operations* (page 240).

Determine the Name of the Shard to Remove

To determine the name of the shard, connect to a mongos instance with the mongo shell and either:

- Use the listShards command, as in the following:

  ```
  db.adminCommand( { listShards: 1 } )
  ```

- Run either the sh.status() or the db.printShardingStatus() method.

The shards._id field lists the name of each shard.

Remove Chunks from the Shard

From the admin database, run the removeShard command. This begins "draining" chunks from the shard you are removing to other shards in the cluster. For example, for a shard named mongodb0, run:

```
use admin
db.runCommand( { removeShard: "mongodb0" } )
```

This operation returns immediately, with the following response:

```
{
    "msg" : "draining started successfully",
    "state" : "started",
    "shard" : "mongodb0",
```

```
    "ok" : 1
}
```

Depending on your network capacity and the amount of data, this operation can take from a few minutes to several days to complete.

Check the Status of the Migration

To check the progress of the migration at any stage in the process, run `removeShard` from the `admin` database again. For example, for a shard named `mongodb0`, run:

```
use admin
db.runCommand( { removeShard: "mongodb0" } )
```

The command returns output similar to the following:

```
{
     "msg" : "draining ongoing",
    "state" : "ongoing",
    "remaining" : {
        "chunks" : 42,
        "dbs" : 1
    },
    "ok" : 1
}
```

In the output, the `remaining` document displays the remaining number of chunks that MongoDB must migrate to other shards and the number of MongoDB databases that have "primary" status on this shard.

Continue checking the status of the *removeShard* command until the number of chunks remaining is `0`. Always run the command on the `admin` database. If you are on a database other than `admin`, you can use `sh._adminCommand` to run the command on `admin`.

Move Unsharded Data

If the shard is the *primary shard* for one or more databases in the cluster, then the shard will have unsharded data. If the shard is not the primary shard for any databases, skip to the next task, *Finalize the Migration* (page 245).

In a cluster, a database with unsharded collections stores those collections only on a single shard. That shard becomes the primary shard for that database. (Different databases in a cluster can have different primary shards.)

Warning: Do not perform this procedure until you have finished draining the shard.

1. To determine if the shard you are removing is the primary shard for any of the cluster's databases, issue one of the following methods:

   - `sh.status()`
   - `db.printShardingStatus()`

   In the resulting document, the `databases` field lists each database and its primary shard. For example, the following `database` field shows that the `products` database uses `mongodb0` as the primary shard:

   ```
   {  "_id" : "products",  "partitioned" : true,  "primary" : "mongodb0" }
   ```

2. To move a database to another shard, use the `movePrimary` command. For example, to migrate all remaining unsharded data from `mongodb0` to `mongodb1`, issue the following command:

```
db.runCommand( { movePrimary: "products", to: "mongodb1" })
```

This command does not return until MongoDB completes moving all data, which may take a long time. The response from this command will resemble the following:

```
{ "primary" : "mongodb1", "ok" : 1 }
```

Finalize the Migration

To clean up all metadata information and finalize the removal, run `removeShard` again. For example, for a shard named `mongodb0`, run:

```
use admin
db.runCommand( { removeShard: "mongodb0" } )
```

A success message appears at completion:

```
{
    "msg" : "removeshard completed successfully",
    "state" : "completed",
    "shard" : "mongodb0",
    "ok" : 1
}
```

Once the value of the `state` field is "completed", you may safely stop the processes comprising the `mongodb0` shard.

See also:

Backup and Restore Sharded Clusters (page 88)

5.2.3 Sharded Cluster Data Management

The following documents provide information in managing data in sharded clusters.

***Create Chunks in a Sharded Cluster* (page 246)** Create chunks, or *pre-split* empty collection to ensure an even distribution of chunks during data ingestion.

***Split Chunks in a Sharded Cluster* (page 246)** Manually create chunks in a sharded collection.

***Migrate Chunks in a Sharded Cluster* (page 247)** Manually migrate chunks without using the automatic balance process.

***Merge Chunks in a Sharded Cluster* (page 248)** Use the `mergeChunks` to manually combine chunk ranges.

***Modify Chunk Size in a Sharded Cluster* (page 251)** Modify the default chunk size in a sharded collection

***Clear jumbo Flag* (page 252)** Clear *jumbo* flag from a shard.

***Manage Shard Tags* (page 254)** Use tags to associate specific ranges of shard key values with specific shards.

***Enforce Unique Keys for Sharded Collections* (page 256)** Ensure that a field is always unique in all collections in a sharded cluster.

***Shard GridFS Data Store* (page 258)** Choose whether to shard GridFS data in a sharded collection.

Create Chunks in a Sharded Cluster

Pre-splitting the chunk ranges in an empty sharded collection allows clients to insert data into an already partitioned collection. In most situations a *sharded cluster* will create and distribute chunks automatically without user intervention. However, in a limited number of cases, MongoDB cannot create enough chunks or distribute data fast enough to support required throughput. For example:

- If you want to partition an existing data collection that resides on a single shard.
- If you want to ingest a large volume of data into a cluster that isn't balanced, or where the ingestion of data will lead to data imbalance. For example, monotonically increasing or decreasing shard keys insert all data into a single chunk.

These operations are resource intensive for several reasons:

- Chunk migration requires copying all the data in the chunk from one shard to another.
- MongoDB can migrate only a single chunk at a time.
- MongoDB creates splits only after an insert operation.

Warning: Only pre-split an empty collection. If a collection already has data, MongoDB automatically splits the collection's data when you enable sharding for the collection. Subsequent attempts to manually create splits can lead to unpredictable chunk ranges and sizes as well as inefficient or ineffective balancing behavior.

To create chunks manually, use the following procedure:

1. Split empty chunks in your collection by manually performing the `split` command on chunks.

Example

To create chunks for documents in the `myapp.users` collection using the `email` field as the *shard key*, use the following operation in the `mongo` shell:

```
for ( var x=97; x<97+26; x++ ){
  for( var y=97; y<97+26; y+=6 ) {
    var prefix = String.fromCharCode(x) + String.fromCharCode(y);
    db.runCommand( { split : "myapp.users" , middle : { email : prefix } } );
  }
}
```

This assumes a collection size of 100 million documents.

For information on the balancer and automatic distribution of chunks across shards, see *sharding-balancing-internals* and *sharding-chunk-migration*. For information on manually migrating chunks, see *Migrate Chunks in a Sharded Cluster* (page 247).

Split Chunks in a Sharded Cluster

Normally, MongoDB splits a *chunk* after an insert if the chunk exceeds the maximum *chunk size*. However, you may want to split chunks manually if:

- you have a large amount of data in your cluster and very few *chunks*, as is the case after deploying a cluster using existing data.
- you expect to add a large amount of data that would initially reside in a single chunk or shard. For example, you plan to insert a large amount of data with *shard key* values between `300` and `400`, *but* all values of your shard keys are between `250` and `500` are in a single chunk.

Note: New in version 2.6: MongoDB provides the `mergeChunks` command to combine contiguous chunk ranges into a single chunk. See *Merge Chunks in a Sharded Cluster* (page 248) for more information.

The *balancer* may migrate recently split chunks to a new shard immediately if `mongos` predicts future insertions will benefit from the move. The balancer does not distinguish between chunks split manually and those split automatically by the system.

Warning: Be careful when splitting data in a sharded collection to create new chunks. When you shard a collection that has existing data, MongoDB automatically creates chunks to evenly distribute the collection. To split data effectively in a sharded cluster you must consider the number of documents in a chunk and the average document size to create a uniform chunk size. When chunks have irregular sizes, shards may have an equal number of chunks but have very different data sizes. Avoid creating splits that lead to a collection with differently sized chunks.

Use `sh.status()` to determine the current chunk ranges across the cluster.

To split chunks manually, use the `split` command with either fields `middle` or `find`. The `mongo` shell provides the helper methods `sh.splitFind()` and `sh.splitAt()`.

`splitFind()` splits the chunk that contains the *first* document returned that matches this query into two equally sized chunks. You must specify the full namespace (i.e. "`<database>.<collection>`") of the sharded collection to `splitFind()`. The query in `splitFind()` does not need to use the shard key, though it nearly always makes sense to do so.

Example

The following command splits the chunk that contains the value of `63109` for the `zipcode` field in the `people` collection of the `records` database:

```
sh.splitFind( "records.people", { "zipcode": "63109" } )
```

Use `splitAt()` to split a chunk in two, using the queried document as the lower bound in the new chunk:

Example

The following command splits the chunk that contains the value of `63109` for the `zipcode` field in the `people` collection of the `records` database.

```
sh.splitAt( "records.people", { "zipcode": "63109" } )
```

Note: `splitAt()` does not necessarily split the chunk into two equally sized chunks. The split occurs at the location of the document matching the query, regardless of where that document is in the chunk.

Migrate Chunks in a Sharded Cluster

In most circumstances, you should let the automatic *balancer* migrate *chunks* between *shards*. However, you may want to migrate chunks manually in a few cases:

- When *pre-splitting* an empty collection, migrate chunks manually to distribute them evenly across the shards. Use pre-splitting in limited situations to support bulk data ingestion.
- If the balancer in an active cluster cannot distribute chunks within the *balancing window* (page 239), then you will have to migrate chunks manually.

To manually migrate chunks, use the moveChunk command. For more information on how the automatic balancer moves chunks between shards, see *sharding-balancing-internals* and *sharding-chunk-migration*.

Example

Migrate a single chunk

The following example assumes that the field username is the *shard key* for a collection named users in the myapp database, and that the value smith exists within the *chunk* to migrate. Migrate the chunk using the following command in the mongo shell.

```
db.adminCommand( { moveChunk : "myapp.users",
                   find : {username : "smith"},
                   to : "mongodb-shard3.example.net" } )
```

This command moves the chunk that includes the shard key value "smith" to the *shard* named mongodb-shard3.example.net. The command will block until the migration is complete.

Tip

To return a list of shards, use the listShards command.

Example

Evenly migrate chunks

To evenly migrate chunks for the myapp.users collection, put each prefix chunk on the next shard from the other and run the following commands in the mongo shell:

```
var shServer = [ "sh0.example.net", "sh1.example.net", "sh2.example.net", "sh3.example.net", "sh4.exa
for ( var x=97; x<97+26; x++ ){
  for( var y=97; y<97+26; y+=6 ) {
    var prefix = String.fromCharCode(x) + String.fromCharCode(y);
    db.adminCommand({moveChunk : "myapp.users", find : {email : prefix}, to : shServer[(y-97)/6]})
  }
}
```

See *Create Chunks in a Sharded Cluster* (page 246) for an introduction to pre-splitting.

New in version 2.2: The moveChunk command has the: _secondaryThrottle parameter. When set to true, MongoDB ensures that changes to shards as part of chunk migrations replicate to *secondaries* throughout the migration operation. For more information, see *Change Replication Behavior for Chunk Migration* (page 237).

Changed in version 2.4: In 2.4, _secondaryThrottle is true by default.

Warning: The moveChunk command may produce the following error message:

```
The collection's metadata lock is already taken.
```

This occurs when clients have too many open *cursors* that access the migrating chunk. You may either wait until the cursors complete their operations or close the cursors manually.

Merge Chunks in a Sharded Cluster

On this page

Overview

The `mergeChunks` command allows you to collapse empty chunks into neighboring chunks on the same shard. A *chunk* is empty if it has no documents associated with its shard key range.

Important: Empty *chunks* can make the *balancer* assess the cluster as properly balanced when it is not.

Empty chunks can occur under various circumstances, including:

- If a *pre-split* (page 246) creates too many chunks, the distribution of data to chunks may be uneven.
- If you delete many documents from a sharded collection, some chunks may no longer contain data.

This tutorial explains how to identify chunks available to merge, and how to merge those chunks with neighboring chunks.

Procedure

Note: Examples in this procedure use a `users` *collection* in the `test` *database*, using the `username` filed as a *shard key*.

Identify Chunk Ranges In the `mongo` shell, identify the *chunk* ranges with the following operation:

```
sh.status()
```

The output of the `sh.status()` will resemble the following:

```
--- Sharding Status ---
sharding version: {
     "_id" : 1,
     "version" : 4,
     "minCompatibleVersion" : 4,
     "currentVersion" : 5,
     "clusterId" : ObjectId("5260032c901f6712dcd8f400")
}
shards:
     {  "_id" : "shard0000",  "host" : "localhost:30000" }
     {  "_id" : "shard0001",  "host" : "localhost:30001" }
  databases:
     {  "_id" : "admin",  "partitioned" : false,  "primary" : "config" }
     {  "_id" : "test",  "partitioned" : true,  "primary" : "shard0001" }
             test.users
                     shard key: { "username" : 1 }
                     chunks:
                             shard0000       7
                             shard0001       7
                     { "username" : { "$minKey" : 1 } } -->> { "username" : "user16643" } on : shard0
                     { "username" : "user16643" } -->> { "username" : "user2329" } on : shard0000 Ti
```

```
                    { "username" : "user2329" } -->> { "username" : "user29937" } on : shard0000 Tir
                    { "username" : "user29937" } -->> { "username" : "user36583" } on : shard0000 T:
                    { "username" : "user36583" } -->> { "username" : "user43229" } on : shard0000 T:
                    { "username" : "user43229" } -->> { "username" : "user49877" } on : shard0000 T:
                    { "username" : "user49877" } -->> { "username" : "user56522" } on : shard0000 T:
                    { "username" : "user56522" } -->> { "username" : "user63169" } on : shard0001 T:
                    { "username" : "user63169" } -->> { "username" : "user69816" } on : shard0001 T:
                    { "username" : "user69816" } -->> { "username" : "user76462" } on : shard0001 T:
                    { "username" : "user76462" } -->> { "username" : "user83108" } on : shard0001 T:
                    { "username" : "user83108" } -->> { "username" : "user89756" } on : shard0001 T:
                    { "username" : "user89756" } -->> { "username" : "user96401" } on : shard0001 T:
                    { "username" : "user96401" } -->> { "username" : { "$maxKey" : 1 } } on : shard(
```

The chunk ranges appear after the chunk counts for each sharded collection, as in the following excerpts:

Chunk counts:

```
chunks:
        shard0000       7
        shard0001       7
```

Chunk range:

```
{ "username" : "user36583" } -->> { "username" : "user43229" } on : shard0000 Timestamp(6, 0)
```

Verify a Chunk is Empty The `mergeChunks` command requires at least one empty input chunk. In the `mongo` shell, check the amount of data in a chunk using an operation that resembles:

```
db.runCommand({
   "dataSize": "test.users",
   "keyPattern": { username: 1 },
   "min": { "username": "user36583" },
   "max": { "username": "user43229" }
})
```

If the input chunk to `dataSize` is empty, `dataSize` produces output similar to:

```
{ "size" : 0, "numObjects" : 0, "millis" : 0, "ok" : 1 }
```

Merge Chunks Merge two contiguous *chunks* on the same *shard*, where at least one of the contains no data, with an operation that resembles the following:

```
db.runCommand( { mergeChunks: "test.users",
                 bounds: [ { "username": "user68982" },
                           { "username": "user95197" } ]
             } )
```

On success, `mergeChunks` produces the following output:

```
{ "ok" : 1 }
```

On any failure condition, `mergeChunks` returns a document where the value of the `ok` field is `0`.

View Merged Chunks Ranges After merging all empty chunks, confirm the new chunk, as follows:

```
sh.status()
```

The output of `sh.status()` should resemble:

```
--- Sharding Status ---
sharding version: {
     "_id" : 1,
     "version" : 4,
     "minCompatibleVersion" : 4,
     "currentVersion" : 5,
     "clusterId" : ObjectId("5260032c901f6712dcd8f400")
}
shards:
     {  "_id" : "shard0000",  "host" : "localhost:30000" }
     {  "_id" : "shard0001",  "host" : "localhost:30001" }
  databases:
     {  "_id" : "admin",  "partitioned" : false,  "primary" : "config" }
     {  "_id" : "test",  "partitioned" : true,  "primary" : "shard0001" }
             test.users
                     shard key: { "username" : 1 }
                     chunks:
                             shard0000       2
                             shard0001       2
                     { "username" : { "$minKey" : 1 } } -->> { "username" : "user16643" } on : shard(
                     { "username" : "user16643" } -->> { "username" : "user56522" } on : shard0000 T:
                     { "username" : "user56522" } -->> { "username" : "user96401" } on : shard0001 T:
                     { "username" : "user96401" } -->> { "username" : { "$maxKey" : 1 } } on : shard(
```

Modify Chunk Size in a Sharded Cluster

When the first `mongos` connects to a set of *config servers*, it initializes the sharded cluster with a default chunk size of 64 megabytes. This default chunk size works well for most deployments; however, if you notice that automatic migrations have more I/O than your hardware can handle, you may want to reduce the chunk size. For automatic splits and migrations, a small chunk size leads to more rapid and frequent migrations. The allowed range of the chunk size is between 1 and 1024 megabytes, inclusive.

To modify the chunk size, use the following procedure:

1. Connect to any `mongos` in the cluster using the `mongo` shell.
2. Issue the following command to switch to the *config-database*:

   ```
   use config
   ```

3. Issue the following `save()` operation to store the global chunk size configuration value:

   ```
   db.settings.save( { _id:"chunksize", value: <sizeInMB> } )
   ```

Note: The `chunkSize` and `--chunkSize` options, passed at startup to the `mongos`, **do not** affect the chunk size after you have initialized the cluster.

To avoid confusion, *always* set the chunk size using the above procedure instead of the startup options.

Modifying the chunk size has several limitations:

- Automatic splitting only occurs on insert or update.
- If you lower the chunk size, it may take time for all chunks to split to the new size.
- Splits cannot be undone.

- If you increase the chunk size, existing chunks grow only through insertion or updates until they reach the new size.
- The allowed range of the chunk size is between 1 and 1024 megabytes, inclusive.

Clear `jumbo` Flag

On this page

If MongoDB cannot split a chunk that exceeds the *specified chunk size* or contains a number of documents that exceeds the `max`, MongoDB labels the chunk as *jumbo*.

If the chunk size no longer hits the limits, MongoDB clears the `jumbo` flag for the chunk when the `mongos` reloads or rewrites the chunk metadata.

In cases where you need to clear the flag manually, the following procedures outline the steps to manually clear the `jumbo` flag.

Procedures

Divisible Chunks The preferred way to clear the `jumbo` flag from a chunk is to attempt to split the chunk. If the chunk is divisible, MongoDB removes the flag upon successful split of the chunk.

Step 1: Connect to `mongos`. Connect a `mongo` shell to a `mongos`.

Step 2: Find the `jumbo` Chunk. Run `sh.status(true)` to find the chunk labeled `jumbo`.

```
sh.status(true)
```

For example, the following output from sh.status(true) shows that chunk with shard key range `{ "x" : 2 } -->> { "x" : 4 }` is `jumbo`.

```
--- Sharding Status ---
  sharding version: {
     ...
  }
  shards:
     ...
  databases:
     ...
        test.foo
           shard key: { "x" : 1 }
        chunks:
             shard-b  2
             shard-a  2
        { "x" : { "$minKey" : 1 } } -->> { "x" : 1 } on : shard-b Timestamp(2, 0)
        { "x" : 1 } -->> { "x" : 2 } on : shard-a Timestamp(3, 1)
        { "x" : 2 } -->> { "x" : 4 } on : shard-a Timestamp(2, 2) jumbo
        { "x" : 4 } -->> { "x" : { "$maxKey" : 1 } } on : shard-b Timestamp(3, 0)
```

Step 3: Split the `jumbo` Chunk. Use either `sh.splitAt()` or `sh.splitFind()` to split the `jumbo` chunk.

```
sh.splitAt( "test.foo", { x: 3 })
```

MongoDB removes the `jumbo` flag upon successful split of the chunk.

Indivisible Chunks In some instances, MongoDB cannot split the no-longer `jumbo` chunk, such as a chunk with a range of single shard key value, and the preferred method to clear the flag is not applicable. In such cases, you can clear the flag using the following steps.

Important: Only use this method if the *preferred method* (page 252) is *not* applicable.

Before modifying the `config database`, *always* back up the config database.

If you clear the `jumbo` flag for a chunk that still exceeds the chunk size and/or the document number limit, MongoDB will re-label the chunk as `jumbo` when MongoDB tries to move the chunk.

Step 1: Stop the balancer. Disable the cluster balancer process temporarily, following the steps outlined in *Disable the Balancer* (page 240).

Step 2: Create a backup of `config` database. Use `mongodump` against a config server to create a backup of the `config` database. For example:

```
mongodump --db config --port <config server port> --out <output file>
```

Step 3: Connect to `mongos`. Connect a `mongo` shell to a `mongos`.

Step 4: Find the `jumbo` Chunk. Run `sh.status(true)` to find the chunk labeled `jumbo`.

```
sh.status(true)
```

For example, the following output from sh.status(true) shows that chunk with shard key range `{ "x" : 2 } -->> { "x" : 3 }` is `jumbo`.

```
--- Sharding Status ---
  sharding version: {
     ...
  }
  shards:
     ...
  databases:
     ...
        test.foo
           shard key: { "x" : 1 }
        chunks:
             shard-b  2
             shard-a  2
        { "x" : { "$minKey" : 1 } } -->> { "x" : 1 } on : shard-b Timestamp(2, 0)
        { "x" : 1 } -->> { "x" : 2 } on : shard-a Timestamp(3, 1)
        { "x" : 2 } -->> { "x" : 3 } on : shard-a Timestamp(2, 2) jumbo
        { "x" : 3 } -->> { "x" : { "$maxKey" : 1 } } on : shard-b Timestamp(3, 0)
```

Step 5: Update chunks collection. In the chunks collection of the config database, unset the jumbo flag for the chunk. For example,

```
db.getSiblingDB("config").chunks.update(
   { ns: "test.foo", min: { x: 2 }, jumbo: true },
   { $unset: { jumbo: "" } }
)
```

Step 6: Restart the balancer. Restart the balancer, following the steps in *Enable the Balancer* (page 241).

Step 7: Optional. Clear current cluster meta information. To ensure that mongos instances update their cluster information cache, run flushRouterConfig in the admin database.

```
db.adminCommand({ flushRouterConfig: 1 } )
```

Manage Shard Tags

On this page

In a sharded cluster, you can use tags to associate specific ranges of a *shard key* with a specific *shard* or subset of shards.

Tag a Shard

Associate tags with a particular shard using the sh.addShardTag() method when connected to a mongos instance. A single shard may have multiple tags, and multiple shards may also have the same tag.

Example

The following example adds the tag NYC to two shards, and the tags SFO and NRT to a third shard:

```
sh.addShardTag("shard0000", "NYC")
sh.addShardTag("shard0001", "NYC")
sh.addShardTag("shard0002", "SFO")
sh.addShardTag("shard0002", "NRT")
```

You may remove tags from a particular shard using the sh.removeShardTag() method when connected to a mongos instance, as in the following example, which removes the NRT tag from a shard:

```
sh.removeShardTag("shard0002", "NRT")
```

Tag a Shard Key Range

To assign a tag to a range of shard keys use the sh.addTagRange() method when connected to a mongos instance. Any given shard key range may only have *one* assigned tag. You cannot overlap defined ranges, or tag the same range more than once.

Example

Given a collection named `users` in the `records` database, sharded by the `zipcode` field. The following operations assign:

- two ranges of zip codes in Manhattan and Brooklyn the `NYC` tag
- one range of zip codes in San Francisco the `SFO` tag

```
sh.addTagRange("records.users", { zipcode: "10001" }, { zipcode: "10281" }, "NYC")
sh.addTagRange("records.users", { zipcode: "11201" }, { zipcode: "11240" }, "NYC")
sh.addTagRange("records.users", { zipcode: "94102" }, { zipcode: "94135" }, "SFO")
```

Note: Shard ranges are always inclusive of the lower value and exclusive of the upper boundary.

Remove a Tag From a Shard Key Range

The `mongod` does not provide a helper for removing a tag range. You may delete tag assignment from a shard key range by removing the corresponding document from the `tags` collection of the `config` database.

Each document in the `tags` holds the *namespace* of the sharded collection and a minimum shard key value.

Example

The following example removes the `NYC` tag assignment for the range of zip codes within Manhattan:

```
use config
db.tags.remove({ _id: { ns: "records.users", min: { zipcode: "10001" }}, tag: "NYC" })
```

View Existing Shard Tags

The output from `sh.status()` lists tags associated with a shard, if any, for each shard. A shard's tags exist in the shard's document in the `shards` collection of the `config` database. To return all shards with a specific tag, use a sequence of operations that resemble the following, which will return only those shards tagged with `NYC`:

```
use config
db.shards.find({ tags: "NYC" })
```

You can find tag ranges for all *namespaces* in the `tags` collection of the `config` database. The output of `sh.status()` displays all tag ranges. To return all shard key ranges tagged with `NYC`, use the following sequence of operations:

```
use config
db.tags.find({ tags: "NYC" })
```

Additional Resource

- Whitepaper: MongoDB Multi-Data Center Deployments[9]
- Webinar: Multi-Data Center Deployment[10]

[9] http://www.mongodb.com/lp/white-paper/multi-dc?jmp=docs
[10] https://www.mongodb.com/presentations/webinar-multi-data-center-deployment?jmp=docs

Enforce Unique Keys for Sharded Collections

On this page

Overview

The unique constraint on indexes ensures that only one document can have a value for a field in a *collection*. For *sharded collections these unique indexes cannot enforce uniqueness* because insert and indexing operations are local to each shard.

MongoDB does not support creating new unique indexes in sharded collections and will not allow you to shard collections with unique indexes on fields other than the _id field.

If you need to ensure that a field is always unique in a sharded collection, there are three options:

1. Enforce uniqueness of the *shard key*.

   MongoDB *can* enforce uniqueness for the *shard key*. For compound shard keys, MongoDB will enforce uniqueness on the *entire* key combination, and not for a specific component of the shard key.

   You cannot specify a unique constraint on a *hashed index*.

2. Use a secondary collection to enforce uniqueness.

   Create a minimal collection that only contains the unique field and a reference to a document in the main collection. If you always insert into a secondary collection *before* inserting to the main collection, MongoDB will produce an error if you attempt to use a duplicate key.

   If you have a small data set, you may not need to shard this collection and you can create multiple unique indexes. Otherwise you can shard on a single unique key.

3. Use guaranteed unique identifiers.

   Universally unique identifiers (i.e. UUID) like the ObjectId are guaranteed to be unique.

Procedures

Unique Constraints on the Shard Key

Process To shard a collection using the unique constraint, specify the shardCollection command in the following form:

```
db.runCommand( { shardCollection : "test.users" , key : { email : 1 } , unique : true } );
```

Remember that the _id field index is always unique. By default, MongoDB inserts an ObjectId into the _id field. However, you can manually insert your own value into the _id field and use this as the shard key. To use the _id field as the shard key, use the following operation:

```
db.runCommand( { shardCollection : "test.users" } )
```

Limitations

- You can only enforce uniqueness on one single field in the collection using this method.
- If you use a compound shard key, you can only enforce uniqueness on the *combination* of component keys in the shard key.

In most cases, the best shard keys are compound keys that include elements that permit *write scaling* and *query isolation*, as well as *high cardinality* (page 217). These ideal shard keys are not often the same keys that require uniqueness and enforcing unique values in these collections requires a different approach.

Unique Constraints on Arbitrary Fields If you cannot use a unique field as the shard key or if you need to enforce uniqueness over multiple fields, you must create another *collection* to act as a "proxy collection". This collection must contain both a reference to the original document (i.e. its `ObjectId`) and the unique key.

If you must shard this "proxy" collection, then shard on the unique key using the *above procedure* (page 256); otherwise, you can simply create multiple unique indexes on the collection.

Process Consider the following for the "proxy collection:"

```
{
  "_id" : ObjectId("...")
  "email" ": "..."
}
```

The `_id` field holds the `ObjectId` of the *document* it reflects, and the `email` field is the field on which you want to ensure uniqueness.

To shard this collection, use the following operation using the `email` field as the *shard key*:

```
db.runCommand( { shardCollection : "records.proxy" ,
                 key : { email : 1 } ,
                 unique : true } );
```

If you do not need to shard the proxy collection, use the following command to create a unique index on the `email` field:

```
db.proxy.createIndex( { "email" : 1 }, { unique : true } )
```

You may create multiple unique indexes on this collection if you do not plan to shard the `proxy` collection.

To insert documents, use the following procedure in the *JavaScript shell*:

```
db = db.getSiblingDB('records');

var primary_id = ObjectId();

db.proxy.insert({
   "_id" : primary_id
   "email" : "example@example.net"
})

// if: the above operation returns successfully,
// then continue:

db.information.insert({
   "_id" : primary_id
   "email": "example@example.net"
   // additional information...
})
```

You must insert a document into the `proxy` collection first. If this operation succeeds, the `email` field is unique, and you may continue by inserting the actual document into the `information` collection.

See

The full documentation of: `createIndex()` and `shardCollection`.

Considerations

- Your application must catch errors when inserting documents into the "proxy" collection and must enforce consistency between the two collections.
- If the proxy collection requires sharding, you must shard on the single field on which you want to enforce uniqueness.
- To enforce uniqueness on more than one field using sharded proxy collections, you must have *one* proxy collection for *every* field for which to enforce uniqueness. If you create multiple unique indexes on a single proxy collection, you will *not* be able to shard proxy collections.

Use Guaranteed Unique Identifier The best way to ensure a field has unique values is to generate universally unique identifiers (UUID,) such as MongoDB's '`ObjectId` values.

This approach is particularly useful for the``_id`` field, which *must* be unique: for collections where you are *not* sharding by the `_id` field the application is responsible for ensuring that the `_id` field is unique.

Shard GridFS Data Store

On this page

When sharding a *GridFS* store, consider the following:

`files` Collection

Most deployments will not need to shard the `files` collection. The `files` collection is typically small, and only contains metadata. None of the required keys for GridFS lend themselves to an even distribution in a sharded situation. If you *must* shard the `files` collection, use the `_id` field possibly in combination with an application field.

Leaving `files` unsharded means that all the file metadata documents live on one shard. For production GridFS stores you *must* store the `files` collection on a replica set.

`chunks` Collection

To shard the `chunks` collection by `{ files_id : 1 , n : 1 }`, issue commands similar to the following:

```
db.fs.chunks.createIndex( { files_id : 1 , n : 1 } )

db.runCommand( { shardCollection : "test.fs.chunks" , key : { files_id : 1 , n : 1 } } )
```

You may also want to shard using just the `file_id` field, as in the following operation:

```
db.runCommand( { shardCollection : "test.fs.chunks" , key : {  files_id : 1 } } )
```

Important: `{ files_id : 1 , n : 1 }` and `{ files_id : 1 }` are the **only** supported shard keys for the `chunks` collection of a GridFS store.

Note: Changed in version 2.2.

Before 2.2, you had to create an additional index on `files_id` to shard using *only* this field.

The default `files_id` value is an *ObjectId*, as a result the values of `files_id` are always ascending, and applications will insert all new GridFS data to a single chunk and shard. If your write load is too high for a single server to handle, consider a different shard key or use a different value for `_id` in the `files` collection.

5.2.4 Troubleshoot Sharded Clusters

On this page

This section describes common strategies for troubleshooting *sharded cluster* deployments.

Config Database String Error

Start all `mongos` instances in a sharded cluster with an identical `configDB` string. If a `mongos` instance tries to connect to the sharded cluster with a `configDB` string that does not *exactly* match the string used by the other `mongos` instances, including the order of the hosts, the following errors occur:

```
could not initialize sharding on connection
```

And:

```
mongos specified a different config database string
```

To solve the issue, restart the `mongos` with the correct string.

Cursor Fails Because of Stale Config Data

A query returns the following warning when one or more of the `mongos` instances has not yet updated its cache of the cluster's metadata from the *config database*:

```
could not initialize cursor across all shards because : stale config detected
```

This warning *should* not propagate back to your application. The warning will repeat until all the `mongos` instances refresh their caches. To force an instance to refresh its cache, run the `flushRouterConfig` command.

Avoid Downtime when Moving Config Servers

Use CNAMEs to identify your config servers to the cluster so that you can rename and renumber your config servers without downtime.