
MongoDB Documentation

Release 2.4.8

MongoDB Documentation Project

November 01, 2013

1	Install MongoDB	3
1.1	Installation Guides	3
1.2	First Steps with MongoDB	18
2	MongoDB CRUD Operations	25
2.1	MongoDB CRUD Introduction	25
2.2	MongoDB CRUD Concepts	27
2.3	MongoDB CRUD Tutorials	55
2.4	MongoDB CRUD Reference	80
3	Data Models	95
3.1	Data Modeling Introduction	95
3.2	Data Modeling Concepts	97
3.3	Data Model Examples and Patterns	104
3.4	Data Model Reference	119
4	Administration	133
4.1	Administration Concepts	133
4.2	Administration Tutorials	167
4.3	Administration Reference	219
5	Security	233
5.1	Security Introduction	233
5.2	Security Concepts	235
5.3	Security Tutorials	240
5.4	Security Reference	263
6	Aggregation	273
6.1	Aggregation Introduction	273
6.2	Aggregation Concepts	277
6.3	Aggregation Examples	288
6.4	Aggregation Reference	304
7	Indexes	309
7.1	Index Introduction	309
7.2	Index Concepts	314
7.3	Indexing Tutorials	334
7.4	Indexing Reference	370

8 Replication	373
8.1 Replication Introduction	373
8.2 Replication Concepts	377
8.3 Replica Set Tutorials	415
8.4 Replication Reference	462
9 Sharding	487
9.1 Sharding Introduction	487
9.2 Sharding Concepts	492
9.3 Sharded Cluster Tutorials	513
9.4 Sharding Reference	554
10 Frequently Asked Questions	571
10.1 FAQ: MongoDB Fundamentals	571
10.2 FAQ: MongoDB for Application Developers	574
10.3 FAQ: The mongo Shell	584
10.4 FAQ: Concurrency	586
10.5 FAQ: Sharding with MongoDB	590
10.6 FAQ: Replica Sets and Replication in MongoDB	596
10.7 FAQ: MongoDB Storage	599
10.8 FAQ: Indexes	603
10.9 FAQ: MongoDB Diagnostics	606
11 Release Notes	611
11.1 Current Stable Release	611
11.2 Previous Stable Releases	629
11.3 Other MongoDB Release Notes	654
11.4 MongoDB Version Numbers	655
12 About MongoDB Documentation	657
12.1 License	657
12.2 Editions	657
12.3 Version and Revisions	658
12.4 Report an Issue or Make a Change Request	658
12.5 Contribute to the Documentation	658
Index	673

See [About MongoDB Documentation](#) (page 657) for more information about the MongoDB Documentation project, this Manual and additional editions of this text.

Note: This version of the PDF does *not* include the reference section, see [MongoDB Reference Manual](#)¹ for a PDF edition of all MongoDB Reference Material.

¹<http://docs.mongodb.org/v2.4/MongoDB-reference-manual.pdf>

Install MongoDB

MongoDB runs on most platforms and supports both 32-bit and 64-bit architectures.

1.1 Installation Guides

See

Release Notes (page 611) for information about specific releases of MongoDB.

1.1.1 Linux

Install on Linux (page 3)

Install on Linux

These documents provide instructions to install MongoDB for various Linux systems.

Recommended

For easy installation, MongoDB provides packages for popular Linux distributions. The following guides detail the installation process for these systems:

Install on Red Hat Enterprise Linux (page 4) Install MongoDB on Red Hat Enterprise, CentOS, Fedora and related Linux systems using .rpm packages.

Install on Ubuntu (page 6) Install MongoDB on Ubuntu Linux systems using .deb packages.

Install on Debian (page 7) Install MongoDB on Debian systems using .deb packages.

For systems without supported packages, refer to the Manual Installation tutorial.

Manual Installation

Although packages are the preferred installation method, for Linux systems without supported packages, see the following guide:

Install on Other Linux Systems (page 9) Install MongoDB on other Linux systems from the MongoDB archives.

Install MongoDB on Red Hat Enterprise, CentOS, or Fedora This tutorial outlines the steps to install *MongoDB* on Red Hat Enterprise Linux, CentOS Linux, Fedora Linux and related systems. The tutorial uses .rpm packages to install. While some of these distributions include their own MongoDB packages, the official MongoDB packages are generally more up to date.

Packages The MongoDB downloads repository contains two packages:

- mongo-10gen-server

This package contains the mongod and mongos daemons from the latest **stable** release and associated configuration and init scripts. Additionally, you can use this package to *install daemons from a previous release* (page 4) of MongoDB.

- mongo-10gen

This package contains all MongoDB tools from the latest **stable** release. Additionally, you can use this package to *install tools from a previous release* (page 4) of MongoDB. Install this package on all production MongoDB hosts and optionally on other systems from which you may need to administer MongoDB systems.

Install MongoDB

Configure Package Management System (YUM) Create a /etc/yum.repos.d/mongodb.repo file to hold the following configuration information for the MongoDB repository:

Tip

For production deployments, always run MongoDB on 64-bit systems.

If you are running a 64-bit system, use the following configuration:

```
[mongodb]
name=MongoDB Repository
baseurl=http://downloads-distro.mongodb.org/repo/redhat/os/x86_64/
gpgcheck=0
enabled=1
```

If you are running a 32-bit system, which is not recommended for production deployments, use the following configuration:

```
[mongodb]
name=MongoDB Repository
baseurl=http://downloads-distro.mongodb.org/repo/redhat/os/i686/
gpgcheck=0
enabled=1
```

Install Packages Issue the following command (as root or with sudo) to install the latest stable version of MongoDB and the associated tools:

```
yum install mongo-10gen mongo-10gen-server
```

When this command completes, you have successfully installed MongoDB!

Manage Installed Versions You can use the mongo-10gen and mongo-10gen-server packages to install previous releases of MongoDB. To install a specific release, append the version number, as in the following example:

```
yum install mongo-10gen-2.2.3 mongo-10gen-server-2.2.3
```

This installs the `mongo-10gen` and `mongo-10gen-server` packages with the 2.2.3 release. You can specify any available version of MongoDB; however `yum` **will** upgrade the `mongo-10gen` and `mongo-10gen-server` packages when a newer version becomes available. Use the following *pinning* procedure to prevent unintended upgrades.

To pin a package, add the following line to your `/etc/yum.conf` file:

```
exclude=mongo-10gen,mongo-10gen-server
```

Control Scripts

Warning: With the introduction of `systemd` in Fedora 15, the control scripts included in the packages available in the MongoDB downloads repository are not compatible with Fedora systems. A correction is forthcoming, see [SERVER-7285^a](#) for more information, and in the mean time use your own control scripts *or* install using the procedure outlined in [Install MongoDB on Linux Systems](#) (page 9).

^a<https://jira.mongodb.org/browse/SERVER-7285>

The packages include various *control scripts*, including the init script `/etc/rc.d/init.d/mongod`. These packages configure MongoDB using the `/etc/mongod.conf` file in conjunction with the control scripts.

As of version 2.4.8, there are no control scripts for `mongos`. `mongos` is only used in [sharding deployments](#) (page 492). You can use the `mongod` init script to derive your own `mongos` control script.

Run MongoDB

Important: You must SELinux to allow MongoDB to start on Fedora systems. Administrators have two options:

- enable access to the relevant ports (e.g. 27017) for SELinux. See [Configuration Options](#) (page 238) for more information on MongoDB's *default ports* (page 270).
- disable SELinux entirely. This requires a system reboot and may have larger implications for your deployment.

Start MongoDB The MongoDB instance stores its data files in the `/var/lib/mongo` and its log files in `/var/log/mongo`, and run using the `mongod` user account. If you change the user that runs the MongoDB process, you **must** modify the access control rights to the `/var/lib/mongo` and `/var/log/mongo` directories.

Start the `mongod` process by issuing the following command (as root or with `sudo`):

```
service mongod start
```

You can verify that the `mongod` process has started successfully by checking the contents of the log file at `/var/log/mongo/mongod.log`.

You may optionally ensure that MongoDB will start following a system reboot by issuing the following command (with root privileges):

```
chkconfig mongod on
```

Stop MongoDB Stop the `mongod` process by issuing the following command (as root or with `sudo`):

```
service mongod stop
```

Restart MongoDB You can restart the `mongod` process by issuing the following command (as root or with `sudo`):

```
service mongod restart
```

Follow the state of this process by watching the output in the `/var/log/mongo/mongod.log` file to watch for errors or important messages from the server.

Install MongoDB on Ubuntu This tutorial outlines the steps to install *MongoDB* on Ubuntu Linux systems. The tutorial uses `.deb` packages to install. Although Ubuntu include its own MongoDB packages, the official MongoDB packages are generally more up to date.

Note: If you use an older Ubuntu that does **not** use Upstart, (i.e. any version before 9.10 “Karmic”) please follow the instructions on the [Install MongoDB on Debian](#) (page 7) tutorial.

Package Options The MongoDB downloads repository provides the `mongodb-10gen` package, which contains the latest **stable** release. Additionally you can [install previous releases](#) (page 6) of MongoDB.

You cannot install this package concurrently with the `mongodb`, `mongodb-server`, or `mongodb-clients` packages provided by Ubuntu.

Install MongoDB

Configure Package Management System (APT) The Ubuntu package management tool (i.e. `dpkg` and `apt`) ensure package consistency and authenticity by requiring that distributors sign packages with GPG keys. Issue the following command to import the [MongoDB public GPG Key](#)¹:

```
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 7F0CEB10
```

Create a `/etc/apt/sources.list.d/mongodb.list` file using the following command.

```
echo 'deb http://downloads-distro.mongodb.org/repo/ubuntu-upstart dist 10gen' | sudo tee /etc/apt/sou
```

Now issue the following command to reload your repository:

```
sudo apt-get update
```

Install Packages Issue the following command to install the latest stable version of MongoDB:

```
sudo apt-get install mongodb-10gen
```

When this command completes, you have successfully installed MongoDB! Continue for configuration and start-up suggestions.

Manage Installed Versions You can use the `mongodb-10gen` package to install previous versions of MongoDB. To install a specific release, append the version number to the package name, as in the following example:

```
apt-get install mongodb-10gen=2.2.3
```

This will install the `2.2.3` release of MongoDB. You can specify any available version of MongoDB; however `apt-get` **will** upgrade the `mongodb-10gen` package when a newer version becomes available. Use the following *pinning* procedure to prevent unintended upgrades.

¹<http://docs.mongodb.org/10gen-gpg-key.asc>

To pin a package, issue the following command at the system prompt to *pin* the version of MongoDB at the currently installed version:

```
echo "mongodb-10gen hold" | sudo dpkg --set-selections
```

Control Scripts The packages include various *control scripts*, including the init script `/etc/rc.d/init.d/mongod`. These packages configure MongoDB using the `/etc/mongod.conf` file in conjunction with the control scripts.

As of version 2.4.8, there are no control scripts for `mongos`. `mongos` is only used in [sharding deployments](#) (page 492). You can use the `mongod` init script to derive your own `mongos` control script.

Run MongoDB The MongoDB instance stores its data files in the `/var/lib/mongo` and its log files in `/var/log/mongo`, and run using the `mongod` user account. If you change the user that runs the MongoDB process, you **must** modify the access control rights to the `/var/lib/mongo` and `/var/log/mongo` directories.

Start MongoDB You can start the `mongod` process by issuing the following command:

```
sudo service mongodb start
```

You can verify that `mongod` has started successfully by checking the contents of the log file at `/var/log/mongodb/mongod.log`.

Stop MongoDB As needed, you may stop the `mongod` process by issuing the following command:

```
sudo service mongodb stop
```

Restart MongoDB You may restart the `mongod` process by issuing the following command:

```
sudo service mongodb restart
```

Install MongoDB on Debian This tutorial outlines the steps to install *MongoDB* on Debian systems. The tutorial uses `.deb` packages to install. While some Debian distributions include their own MongoDB packages, the official MongoDB packages are generally more up to date.

Note: This tutorial applies to both Debian systems and versions of Ubuntu Linux prior to 9.10 “Karmic” which do not use Upstart. Other Ubuntu users will want to follow the [Install MongoDB on Ubuntu](#) (page 6) tutorial.

Package Options The downloads repository provides the `mongodb-10gen` package, which contains the latest **stable** release. Additionally you can [install previous releases](#) (page 8) of MongoDB.

You cannot install this package concurrently with the `mongodb`, `mongodb-server`, or `mongodb-clients` packages that your release of Debian may include.

Install MongoDB

Configure Package Management System (APT) The Debian package management tools (i.e. `dpkg` and `apt`) ensure package consistency and authenticity by requiring that distributors sign packages with GPG keys.

1. Import the MongoDB public GPG Key².

```
sudo apt-key adv --keyserver keyserver.ubuntu.com --recv 7F0CEB10
```

2. Create a `/etc/apt/sources.list.d/mongodb.list` file.

```
echo 'deb http://downloads-distro.mongodb.org/repo/debian-sysvinit dist 10gen' | sudo tee /etc/a
```

3. Reload your repository.

```
sudo apt-get update
```

Install Packages Issue the following command to install the latest stable version of MongoDB:

```
sudo apt-get install mongodb-10gen
```

When this command completes, you have successfully installed MongoDB!

Manage Installed Versions You can use the `mongodb-10gen` package to install previous versions of MongoDB. To install a specific release, append the version number to the package name, as in the following example:

```
apt-get install mongodb-10gen=2.2.3
```

This will install the `2.2.3` release of MongoDB. You can specify any available version of MongoDB; however `apt-get` **will** upgrade the `mongodb-10gen` package when a newer version becomes available. Use the following *pinning* procedure to prevent unintended upgrades.

To pin a package, issue the following command at the system prompt to *pin* the version of MongoDB at the currently installed version:

```
echo "mongodb-10gen hold" | sudo dpkg --set-selections
```

Control Scripts The packages include various *control scripts*, including the `init` script `/etc/rc.d/init.d/mongod`. These packages configure MongoDB using the `/etc/mongod.conf` file in conjunction with the control scripts.

As of version 2.4.8, there are no control scripts for `mongos`. `mongos` is only used in *sharding deployments* (page 492). You can use the `mongod` init script to derive your own `mongos` control script.

Run MongoDB The MongoDB instance stores its data files in the `/var/lib/mongo` and its log files in `/var/log/mongo`, and run using the `mongod` user account. If you change the user that runs the MongoDB process, you **must** modify the access control rights to the `/var/lib/mongo` and `/var/log/mongo` directories.

Start MongoDB Issue the following command to start `mongod`:

```
sudo /etc/init.d/mongodb start
```

You can verify that `mongod` has started successfully by checking the contents of the log file at `/var/log/mongodb/mongod.log`.

²<http://docs.mongodb.org/10gen-gpg-key.asc>

Stop MongoDB Issue the following command to stop mongod:

```
sudo /etc/init.d/mongodb stop
```

Restart MongoDB Issue the following command to restart mongod:

```
sudo /etc/init.d/mongodb restart
```

Install MongoDB on Linux Systems Compiled versions of MongoDB for Linux provide a simple option for installing MongoDB for other Linux systems without supported packages.

Installation Process MongoDB provides archives for both 64-bit and 32-bit Linux. Follow the installation procedure appropriate for your system.

Install for 64-bit Linux

1. In a system shell, download the latest release for 64-bit ³.

```
curl http://downloads.mongodb.org/linux/mongodb-linux-x86_64-2.4.8.tgz > mongodb-linux-x86_64-2.4.8.tgz
```

2. Extract the files from the downloaded archive.

```
tar -zxvf mongodb-linux-x86_64-2.4.8.tgz
```

3. Optional. Copy the extracted folder into another location, such as `mongodb`.

```
mkdir -p mongodb
cp -R -n mongodb-linux-x86_64-2.4.8/ mongodb
```

4. Optional. To ensure that the downloaded binaries are in your PATH, you can modify your PATH and/or create symbolic links to the MongoDB binaries in your /usr/local/bin directory (/usr/local/bin is already in your PATH). You can find the MongoDB binaries in the bin/ directory within the archive.

Install for 32-bit Linux

1. In a system shell, download the latest release for 32-bit version ¹.

```
curl http://downloads.mongodb.org/linux/mongodb-linux-i686-2.4.8.tgz > mongodb-linux-i686-2.4.8.tgz
```

2. Extract the files from the downloaded archive.

```
tar -zxvf mongodb-linux-i686-2.4.8.tgz
```

3. Optional. Copy the extracted folder into another location, such as `mongodb`.

```
mkdir -p mongodb
cp -R -n mongodb-linux-i686-2.4.8/ mongodb
```

4. Optional. To ensure that the downloaded binaries are in your PATH, you can modify your PATH and/or create symbolic links to the MongoDB binaries in your /usr/local/bin directory (/usr/local/bin is already in your PATH). You can find the MongoDB binaries in the bin/ directory within the archive.

Run MongoDB

³ You can specify a different version to download.

Set Up the Data Directory Before you start mongod for the first time, you will need to create the data directory. By default, mongod writes data to the /data/db directory.

1. To create this directory, use the following command:

```
mkdir -p /data/db
```

2. Ensure that the user that runs the mongod process has read and write permissions to this directory. For example, if you will run the mongod process, change the owner of the /data/db directory:

```
chown mongodb /data/db
```

You can specify an alternate path for data files using the `--dbpath` option to mongod. If you use an alternate location for your data directory, ensure that this user can write to the alternate data directory.

Start MongoDB To start mongod, run the executable mongod at the system prompt.

For example, if your PATH includes the location of the mongod binary, enter mongod at the system prompt.

```
mongod
```

If your PATH does not include the location of the mongod binary, enter the full path to the mongod binary.

Starting mongod without any arguments starts a MongoDB instance that writes data to the /data/db directory. To specify an alternate data directory, start mongod with the `--dbpath` option:

```
mongod --dbpath <some alternate directory>
```

Whether using the default /data/db or an alternate directory, ensure that the user account running mongod has read and write permissions to the directory.

Stop MongoDB To stop the mongod instance, press Control+C in the terminal where the mongod instance is running.

1.1.2 OS X

Install MongoDB on OS X (page 10)

Install MongoDB on OS X

Platform Support

Starting in version 2.4, MongoDB only supports OS X versions 10.6 (Snow Leopard) on Intel x86-64 and later.

MongoDB is available through the popular OS X package manager [Homebrew](#)⁴ or through the MongoDB Download site.

Install MongoDB with Homebrew

Homebrew⁵ ⁶ installs binary packages based on published “formulae”. The following commands will update brew to the latest packages and install MongoDB.

⁴<http://mxcl.github.com/homebrew/>

⁵<http://mxcl.github.com/homebrew/>

⁶ Homebrew requires some initial setup and configuration. This configuration is beyond the scope of this document.

In a terminal shell, update brew to the latest packages and install MongoDB:

```
brew update
brew install mongodb
```

Later, if you need to upgrade MongoDB, run the following sequence of commands to update the MongoDB installation on your system:

```
brew update
brew upgrade mongodb
```

Manual Installation

Download from MongoDB Site

1. In a terminal shell, download the latest release.

```
curl http://downloads.mongodb.org/osx/mongodb-osx-x86_64-2.4.8.tgz > mongodb-osx-x86_64-2.4.8.tgz
```

2. Extract the files from the downloaded archive.

```
tar -zvxf mongodb-osx-x86_64-2.4.8.tgz
```

3. Optional. Copy the extracted folder into another location, such as `mongodb`.

```
mkdir -p mongodb
cp -R -n mongodb-osx-x86_64-2.4.8/ mongodb
```

4. Optional. To ensure that the downloaded binaries are in your PATH, you can modify your PATH and/or create symbolic links to the MongoDB binaries in your /usr/local/bin directory (/usr/local/bin is already in your PATH). You can find the MongoDB binaries in the bin/ directory within the archive.

Run MongoDB

Set Up the Data Directory Before you start mongod for the first time, you will need to create the data directory. By default, mongod writes data to the /data/db/ directory.

1. Create the /data/db/ directory

```
sudo mkdir -p /data/db
```

2. Ensure that the user who runs the mongod process has read and write permissions to this directory. For example, if you will run the mongod process, change the owner of the /data/db/ directory:

```
sudo chown `id -u` /data/db
```

You can specify an alternate path for data files using the `--dbpath` option to mongod. If you use an alternate location for your data directory, ensure that the alternate directory has the appropriate permissions.

Start MongoDB To start mongod, run the executable mongod at the system prompt.

For example, if your PATH includes the location of the mongod binary, enter mongod at the system prompt.

```
mongod
```

If your PATH does not include the location of the `mongod` binary, enter the full path to the `mongod` binary.

The previous command starts a `mongod` instance that writes data to the `/data/db/` directory. To specify an alternate data directory, start `mongod` with the `--dbpath` option:

```
mongod --dbpath <some alternate directory>
```

Whether using the default `/data/db/` or an alternate directory, ensure that the user account running `mongod` has read and write permissions to the directory.

Stop MongoDB To stop the `mongod` instance, press `Control+C` in the terminal where the `mongod` instance is running.

1.1.3 Windows

Install MongoDB on Windows (page 12)

Install MongoDB on Windows

Platform Support

Starting in version 2.2, MongoDB does not support Windows XP. Please use a more recent version of Windows to use more recent releases of MongoDB.

Important: If you are running any edition of Windows Server 2008 R2 or Windows 7, please install a hotfix to resolve an issue with memory mapped files on Windows⁷.

Download MongoDB for Windows

There are three builds of MongoDB for Windows:

- MongoDB for Windows Server 2008 R2 edition (i.e. 2008R2) only runs on Windows Server 2008 R2, Windows 7 64-bit, and newer versions of Windows. This build takes advantage of recent enhancements to the Windows Platform and cannot operate on older versions of Windows.
- MongoDB for Windows 64-bit runs on any 64-bit version of Windows newer than Windows XP, including Windows Server 2008 R2 and Windows 7 64-bit.
- MongoDB for Windows 32-bit runs on any 32-bit version of Windows newer than Windows XP. 32-bit versions of MongoDB are only intended for older systems and for use in testing and development systems. 32-bit versions of MongoDB only support databases smaller than 2GB.

Tip

To find which version of Windows you are running, enter the following command in the *Command Prompt*:

```
wmic os get osarchitecture
```

-
1. Download the latest production release of MongoDB from the [MongoDB downloads page](#)⁸. Ensure you download the correct version of MongoDB for your Windows system. The 64-bit versions of MongoDB will not work with 32-bit Windows.

⁷<http://support.microsoft.com/kb/2731284>

⁸<http://www.mongodb.org/downloads>

2. Extract the downloaded archive.
 - (a) In Windows Explorer, find the MongoDB download file, typically in the default `Downloads` directory.
 - (b) Extract the archive to `C:\` by right clicking on the archive and selecting *Extract All* and browsing to `C:\`.
3. Optional. Move the MongoDB directory to another location. For example, to move the directory to `C:\mongodb` directory:
 - (a) Go *Start Menu > All Programs > Accessories*.
 - (b) Right click *Command Prompt*, and select *Run as Administrator* from the popup menu.
 - (c) In the *Command Prompt*, issue the following commands:

```
cd \
move C:\mongodb-win32-* C:\mongodb
```

Note: MongoDB is self-contained and does not have any other system dependencies. You can run MongoDB from any folder you choose. You may install MongoDB in any directory (e.g. `D:\test\mongodb`)

Run MongoDB

Set Up the Data Directory MongoDB requires a *data folder* to store its files. The default location for the MongoDB data directory is `C:\data\db`. Create this folder using the *Command Prompt*. Go to the `C:\` directory and issue the following command sequence:

```
md data
md data\db
```

You can specify an alternate path for data files using the `--dbpath` option to `mongod.exe`.

Start MongoDB To start MongoDB, execute from the *Command Prompt*:

```
C:\mongodb\bin\mongod.exe
```

This will start the main MongoDB database process. The `waiting for connections` message in the console output indicates that the `mongod.exe` process is running successfully.

Note: Depending on the security level of your system, Windows will issue a *Security Alert* dialog box about blocking “some features” of `C:\mongodb\bin\mongod.exe` from communicating on networks. All users should select *Private Networks*, such as my home or work network and click *Allow access*. For additional information on security and MongoDB, please read the [Security Concepts](#) (page 235) page.

Warning: Do not allow `mongod.exe` to be accessible to public networks without running in “Secure Mode” (i.e. `auth.`) MongoDB is designed to be run in “trusted environments” and the database does not enable authentication or “Secure Mode” by default.

You may specify an alternate path for `\data\db` with the `dbpath` setting for `mongod.exe`, as in the following example:

```
C:\mongodb\bin\mongod.exe --dbpath d:\test\mongodb\data
```

If your path includes spaces, enclose the entire path in double quotations, for example:

```
C:\mongodb\bin\mongod.exe --dbpath "d:\test\mongo db data"
```

Connect to MongoDB Connect to MongoDB using the `mongo.exe` shell. Open another *Command Prompt* and issue the following command:

```
C:\mongodb\bin\mongo.exe
```

Note: Executing the command `start C:\mongodb\bin\mongo.exe` will automatically start the `mongo.exe` shell in a separate *Command Prompt* window.

The `mongo.exe` shell will connect to `mongod.exe` running on the localhost interface and port 27017 by default. At the `mongo.exe` prompt, issue the following two commands to insert a record in the `test` *collection* of the default `test` database and then retrieve that record:

```
db.test.save( { a: 1 } )
db.test.find()
```

See also:

`mongo` and <http://docs.mongodb.org/manual/reference/method>. If you want to develop applications using .NET, see the documentation of [C# and MongoDB⁹](#) for more information.

MongoDB as a Windows Service

New in version 2.0.

You can set up MongoDB as a *Windows Service* so that the database will start automatically following each reboot cycle.

Note: `mongod.exe` added support for running as a Windows service in version 2.0, and `mongos.exe` added support for running as a Windows Service in version 2.1.1.

Configure the System The following steps, although optional, are good practice.

You should specify two options when running MongoDB as a Windows Service: a path for the log output (i.e. `logpath`) and a configuration file.

1. Optional. Create a specific directory for MongoDB log files:

```
md C:\mongodb\log
```

2. Optional. Create a configuration file for the `logpath` option for MongoDB in the *Command Prompt* by issuing this command:

```
echo logpath=C:\mongodb\log\mongo.log > C:\mongodb\mongod.cfg
```

Note: Consider setting the `logappend` option. If you do not, `mongod.exe` will delete the contents of the existing log file when starting.

Changed in version 2.2: The default `logpath` and `logappend` behavior changed in the 2.2 release.

⁹<http://docs.mongodb.org/ecosystem/drivers/csharp>

Install and Run the MongoDB Service Run all of the following commands in *Command Prompt* with “Administrative Privileges:”

1. To install the MongoDB service:

```
C:\mongodb\bin\mongod.exe --config C:\mongodb\mongod.cfg --install
```

Modify the path to the `mongod.cfg` file as needed. For the `--install` option to succeed, you *must* specify a `logpath` setting or the `--logpath` run-time option.

2. To run the MongoDB service:

```
net start MongoDB
```

If you wish to use an alternate path for your `dbpath` specify it in the config file (e.g. `C:\mongodb\mongod.cfg`) on that you specified in the `--install` operation. You may also specify `--dbpath` on the command line; however, always prefer the configuration file.

If you have not set up the data directory, [set up the data directory](#) (page 13) where MongoDB will store its data files. If the `dbpath` directory does not exist, `mongod.exe` will not be able to start. The default value for `dbpath` is `\data\db`.

Stop or Remove the MongoDB Service To stop the MongoDB service:

```
net stop MongoDB
```

To remove the MongoDB service:

```
C:\mongodb\bin\mongod.exe --remove
```

1.1.4 MongoDB Enterprise

[Install MongoDB Enterprise](#) (page 15)

Install MongoDB Enterprise

New in version 2.2.

MongoDB Enterprise¹⁰ is available on four platforms and contains support for several features related to security and monitoring.

Required Packages

Changed in version 2.4.4: MongoDB Enterprise uses Cyrus SASL instead of GNU SASL. Earlier 2.4 Enterprise versions use GNU SASL (`libgsasl`) instead. For required packages for the earlier 2.4 versions, see [Earlier 2.4 Versions](#) (page 16).

To use MongoDB Enterprise, you must install several prerequisites. The names of the packages vary by distribution and are as follows:

- Debian or Ubuntu 12.04 require: `libssl0.9.8`, `snmp`, `snmpd`, `cyrus-sasl2-dbg`, `cyrus-sasl2-mit-dbg`, `libsasl2-2`, `libsasl2-dev`, `libsasl2-modules`, and `libsasl2-modules-gssapi-mit`. Issue a command such as the following to install these packages:

¹⁰<http://www.mongodb.com/products/mongodb-enterprise>

```
sudo apt-get install libssl0.9.8 snmp snmpd cyrus-sasl2-dbg cyrus-sasl2-mit-dbg libsasl2-2 libsasl2-common
```

- CentOS and Red Hat Enterprise Linux 6.x and 5.x, as well as Amazon Linux AMI require: net-snmp, net-snmp-libs, openssl, net-snmp-utils, cyrus-sasl, cyrus-sasl-lib, cyrus-sasl-devel, and cyrus-sasl-gssapi. Issue a command such as the following to install these packages:

```
sudo yum install openssl net-snmp net-snmp-libs net-snmp-utils cyrus-sasl cyrus-sasl-lib cyrus-sasl-devel
```

- SUSE Enterprise Linux requires libopenssl0_9_8, libsnmp15, slessp1-libsnmp15, snmp-mibs, cyrus-sasl, cyrus-sasl-devel, and cyrus-sasl-gssapi. Issue a command such as the following to install these packages:

```
sudo zypper install libopenssl0_9_8 libsnmp15 slessp1-libsnmp15 snmp-mibs cyrus-sasl cyrus-sasl-devel
```

Earlier 2.4 Versions Before version 2.4.4, the 2.4 versions of MongoDB Enterprise use libgsasl¹¹. The required packages for the different distributions are as follows:

- Ubuntu 12.04 requires libssl0.9.8, libgsasl, snmp, and snmpd. Issue a command such as the following to install these packages:

```
sudo apt-get install libssl0.9.8 libgsasl7 snmp snmpd
```

- Red Hat Enterprise Linux 6.x series and Amazon Linux AMI require openssl, libgsasl7, net-snmp, net-snmp-libs, and net-snmp-utils. To download libgsasl you must enable the EPEL repository by issuing the following sequence of commands to add and update the system repositories:

```
sudo rpm -ivh http://download.fedoraproject.org/pub/epel/6/x86_64/epel-release-6-8.noarch.rpm
```

```
sudo yum update -y
```

When you have installed and updated the EPEL repositories, issue the following to install these packages:

```
sudo yum install openssl net-snmp net-snmp-libs net-snmp-utils libgsasl
```

- SUSE Enterprise Linux requires libopenssl0_9_8, libsnmp15, slessp1-libsnmp15, and snmp-mibs. Issue a command such as the following to install these packages:

```
sudo zypper install libopenssl0_9_8 libsnmp15 slessp1-libsnmp15 snmp-mibs
```

Note: Before 2.4.4, MongoDB Enterprise 2.4 for SUSE requires libgsasl¹² which is not available in the default repositories for SUSE.

Install MongoDB Enterprise Binaries

When you have installed the required packages, and downloaded the Enterprise packages¹³ you can install the packages using the same procedure as a standard *installation of MongoDB on Linux Systems* (page 9).

Note: .deb and .rpm packages for Enterprise releases are available for some platforms. You can use these to install MongoDB directly using the dpkg and rpm utilities.

¹¹<http://www.gnu.org/software/gsasl/>

¹²<http://www.gnu.org/software/gsasl/>

¹³<http://www.mongodb.com/products/mongodb-enterprise>

Use the sequence of commands below to download and extract MongoDB Enterprise packages appropriate for your distribution:

Ubuntu 12.04

```
curl http://downloads.10gen.com/linux/mongodb-linux-x86_64-subscription-ubuntu1204-2.4.8.tgz > mongodb-ubuntu1204-2.4.8.tgz
tar -zxvf mongodb-linux-x86_64-subscription-ubuntu1204-2.4.8.tgz
cp -R -n mongodb-linux-x86_64-subscription-ubuntu1204-2.4.8/ mongodb
```

Red Hat Enterprise Linux 6.x

```
curl http://downloads.10gen.com/linux/mongodb-linux-x86_64-subscription-rhel62-2.4.8.tgz > mongodb-rhel62-2.4.8.tgz
tar -zxvf mongodb-linux-x86_64-subscription-rhel62-2.4.8.tgz
cp -R -n mongodb-linux-x86_64-subscription-rhel62-2.4.8/ mongodb
```

Amazon Linux AMI

```
curl http://downloads.10gen.com/linux/mongodb-linux-x86_64-subscription-amzn64-2.4.8.tgz > mongodb-amzn64-2.4.8.tgz
tar -zxvf mongodb-linux-x86_64-subscription-amzn64-2.4.8.tgz
cp -R -n mongodb-linux-x86_64-subscription-amzn64-2.4.8/ mongodb
```

SUSE Enterprise Linux

```
curl http://downloads.10gen.com/linux/mongodb-linux-x86_64-subscription-suse11-2.4.8.tgz > mongodb-suse11-2.4.8.tgz
tar -zxvf mongodb-linux-x86_64-subscription-suse11-2.4.8.tgz
cp -R -n mongodb-linux-x86_64-subscription-suse11-2.4.8/ mongodb
```

Running and Using MongoDB

Note: The Enterprise packages currently include an example SNMP configuration file named `mongod.conf`. This file is not a MongoDB configuration file.

Before you start `mongod` for the first time, you will need to create the data directory. By default, `mongod` writes data to the `/data/db` directory.

1. To create this directory, use the following command:

```
mkdir -p /data/db
```

2. Ensure that the user that runs the `mongod` process has read and write permissions to this directory. For example, if you will run the `mongod` process, change the owner of the `/data/db` directory:

```
chown mongodb /data/db
```

You can specify an alternate path for data files using the `--dbpath` option to `mongod`. If you use an alternate location for your data directory, ensure that this user can write to the alternate data directory.

Start MongoDB To start `mongod`, run the executable `mongod` at the system prompt.

For example, if your `PATH` includes the location of the `mongod` binary, enter `mongod` at the system prompt.

```
mongod
```

If your PATH does not include the location of the `mongod` binary, enter the full path to the `mongod` binary.

Starting `mongod` without any arguments starts a MongoDB instance that writes data to the `/data/db` directory. To specify an alternate data directory, start `mongod` with the `--dbpath` option:

```
mongod --dbpath <some alternate directory>
```

Whether using the default `/data/db` or an alternate directory, ensure that the user account running `mongod` has read and write permissions to the directory.

Stop MongoDB To stop the `mongod` instance, press `Control+C` in the terminal where the `mongod` instance is running.

Further Reading

As you begin to use MongoDB, consider the [Getting Started with MongoDB](#) (page 18) and [MongoDB Tutorials](#) (page 181) resources. To read about features only available in MongoDB Enterprise, consider: [Monitor MongoDB with SNMP](#) (page 174) and [Deploy MongoDB with Kerberos Authentication](#) (page 257).

1.2 First Steps with MongoDB

After you have installed MongoDB, consider the following documents as you begin to learn about MongoDB:

[Getting Started with MongoDB \(page 18\)](#) An introduction to the basic operation and use of MongoDB.

[Generate Test Data \(page 23\)](#) To support initial exploration, generate test data to facilitate testing.

1.2.1 Getting Started with MongoDB

This tutorial addresses the following aspects of MongoDB use:

- Connect to a Database (page 19)
 - Connect to a `mongod` (page 19)
 - Select a Database (page 19)
 - Display `mongo` Help (page 19)
- Create a Collection and Insert Documents (page 20)
- Insert Documents using a For Loop or a JavaScript Function (page 20)
- Working with the Cursor (page 20)
 - Iterate over the Cursor with a Loop (page 21)
 - Use Array Operations with the Cursor (page 21)
 - Query for Specific Documents (page 22)
 - Return a Single Document from a Collection (page 22)
 - Limit the Number of Documents in the Result Set (page 22)
- Next Steps with MongoDB (page 23)

This tutorial provides an introduction to basic database operations using the `mongo` shell. `mongo` is a part of the standard MongoDB distribution and provides a full JavaScript environment with a complete access to the JavaScript language and all standard functions as well as a full database interface for MongoDB. See the [mongo JavaScript API¹⁴](#) documentation and the [mongo shell JavaScript Method Reference](#).

¹⁴<http://api.mongodb.org/js>

The tutorial assumes that you're running MongoDB on a Linux or OS X operating system and that you have a running database server; MongoDB does support Windows and provides a Windows distribution with identical operation. For instructions on installing MongoDB and starting the database server, see the appropriate [installation](#) (page 3) document.

Connect to a Database

In this section, you connect to the database server, which runs as mongod, and begin using the mongo shell to select a logical database within the database instance and access the help text in the mongo shell.

Connect to a mongod

From a system prompt, start mongo by issuing the mongo command, as follows:

```
mongo
```

By default, mongo looks for a database server listening on port 27017 on the localhost interface. To connect to a server on a different port or interface, use the `--port` and `--host` options.

Select a Database

After starting the mongo shell your session will use the test database by default. At any time, issue the following operation at the mongo to report the name of the current database:

```
db
```

1. From the mongo shell, display the list of databases, with the following operation:

```
show dbs
```

2. Switch to a new database named mydb, with the following operation:

```
use mydb
```

3. Confirm that your session has the mydb database as context, by checking the value of the db object, which returns the name of the current database, as follows:

```
db
```

At this point, if you issue the show dbs operation again, it will not include the mydb database. MongoDB will not permanently create a database until you insert data into that database. The [Create a Collection and Insert Documents](#) (page 20) section describes the process for inserting data.

New in version 2.4: show databases also returns a list of databases.

Display mongo Help

At any point, you can access help for the mongo shell using the following operation:

```
help
```

Furthermore, you can append the `.help()` method to some JavaScript methods, any cursor object, as well as the db and db.collection objects to return additional help information.

Create a Collection and Insert Documents

In this section, you insert documents into a new *collection* named `testData` within the new *database* named `mydb`. MongoDB will create a collection implicitly upon its first use. You do not need to create a collection before inserting data. Furthermore, because MongoDB uses *dynamic schemas* (page 572), you also need not specify the structure of your documents before inserting them into the collection.

1. From the `mongo` shell, confirm you are in the `mydb` database by issuing the following:

```
db
```

2. If `mongo` does not return `mydb` for the previous operation, set the context to the `mydb` database, with the following operation:

```
use mydb
```

3. Create two documents named `j` and `k` by using the following sequence of JavaScript operations:

```
j = { name : "mongo" }
k = { x : 3 }
```

4. Insert the `j` and `k` documents into the `testData` collection with the following sequence of operations:

```
db.testData.insert( j )
db.testData.insert( k )
```

When you insert the first document, the `mongod` will create both the `mydb` database and the `testData` collection.

5. Confirm that the `testData` collection exists. Issue the following operation:

```
show collections
```

The `mongo` shell will return the list of the collections in the current (i.e. `mydb`) database. At this point, the only collection is `testData`. All `mongod` databases also have a `system.indexes` (page 223) collection.

6. Confirm that the documents exist in the `testData` collection by issuing a query on the collection using the `find()` method:

```
db.testData.find()
```

This operation returns the following results. The *ObjectId* (page 127) values will be unique:

```
{ "_id" : ObjectId("4c2209f9f3924d31102bd84a"), "name" : "mongo" }
{ "_id" : ObjectId("4c2209fef3924d31102bd84b"), "x" : 3 }
```

All MongoDB documents must have an `_id` field with a unique value. These operations do not explicitly specify a value for the `_id` field, so `mongo` creates a unique *ObjectId* (page 127) value for the field before inserting it into the collection.

Insert Documents using a For Loop or a JavaScript Function

To perform the remaining procedures in this tutorial, first add more documents to your database using one or both of the procedures described in [Generate Test Data](#) (page 23).

Working with the Cursor

When you query a *collection*, MongoDB returns a “cursor” object that contains the results of the query. The `mongo` shell then iterates over the cursor to display the results. Rather than returning all results at once, the shell iterates over

the cursor 20 times to display the first 20 results and then waits for a request to iterate over the remaining results. In the shell, use `enter it` to iterate over the next set of results.

The procedures in this section show other ways to work with a cursor. For comprehensive documentation on cursors, see [crud-read-cursor](#).

Iterate over the Cursor with a Loop

Before using this procedure, make sure to add at least 25 documents to a collection using one of the procedures in [Generate Test Data](#) (page 23). You can name your database and collections anything you choose, but this procedure will assume the database named `test` and a collection named `testData`.

1. In the MongoDB JavaScript shell, query the `testData` collection and assign the resulting cursor object to the `c` variable:

```
var c = db.testData.find()
```

2. Print the full result set by using a `while` loop to iterate over the `c` variable:

```
while ( c.hasNext() ) printjson( c.next() )
```

The `hasNext()` function returns true if the cursor has documents. The `next()` method returns the next document. The `printjson()` method renders the document in a JSON-like format.

The operation displays 20 documents. For example, if the documents have a single field named `x`, the operation displays the field as well as each document's `ObjectId`:

```
{ "_id" : ObjectId("51a7dc7b2cacf40b79990be6"), "x" : 1 }
{ "_id" : ObjectId("51a7dc7b2cacf40b79990be7"), "x" : 2 }
{ "_id" : ObjectId("51a7dc7b2cacf40b79990be8"), "x" : 3 }
{ "_id" : ObjectId("51a7dc7b2cacf40b79990be9"), "x" : 4 }
{ "_id" : ObjectId("51a7dc7b2cacf40b79990bea"), "x" : 5 }
{ "_id" : ObjectId("51a7dc7b2cacf40b79990beb"), "x" : 6 }
{ "_id" : ObjectId("51a7dc7b2cacf40b79990bec"), "x" : 7 }
{ "_id" : ObjectId("51a7dc7b2cacf40b79990bed"), "x" : 8 }
{ "_id" : ObjectId("51a7dc7b2cacf40b79990bee"), "x" : 9 }
{ "_id" : ObjectId("51a7dc7b2cacf40b79990bef"), "x" : 10 }
{ "_id" : ObjectId("51a7dc7b2cacf40b79990bf0"), "x" : 11 }
{ "_id" : ObjectId("51a7dc7b2cacf40b79990bf1"), "x" : 12 }
{ "_id" : ObjectId("51a7dc7b2cacf40b79990bf2"), "x" : 13 }
{ "_id" : ObjectId("51a7dc7b2cacf40b79990bf3"), "x" : 14 }
{ "_id" : ObjectId("51a7dc7b2cacf40b79990bf4"), "x" : 15 }
{ "_id" : ObjectId("51a7dc7b2cacf40b79990bf5"), "x" : 16 }
{ "_id" : ObjectId("51a7dc7b2cacf40b79990bf6"), "x" : 17 }
{ "_id" : ObjectId("51a7dc7b2cacf40b79990bf7"), "x" : 18 }
{ "_id" : ObjectId("51a7dc7b2cacf40b79990bf8"), "x" : 19 }
{ "_id" : ObjectId("51a7dc7b2cacf40b79990bf9"), "x" : 20 }
```

Use Array Operations with the Cursor

The following procedure lets you manipulate a cursor object as if it were an array:

1. In the `mongo` shell, query the `testData` collection and assign the resulting cursor object to the `c` variable:

```
var c = db.testData.find()
```

2. To find the document at the array index 4, use the following operation:

```
printjson( c [ 4 ] )
```

MongoDB returns the following:

```
{ "_id" : ObjectId("51a7dc7b2cacf40b79990bea"), "x" : 5 }
```

When you access documents in a cursor using the array index notation, mongo first calls the `cursor.toArray()` method and loads into RAM all documents returned by the cursor. The index is then applied to the resulting array. This operation iterates the cursor completely and exhausts the cursor.

For very large result sets, mongo may run out of available memory.

For more information on the cursor, see [crud-read-cursor](#).

Query for Specific Documents

MongoDB has a rich query system that allows you to select and filter the documents in a collection along specific fields and values. See [Query Documents](#) (page 57) and [Read Operations](#) (page 29) for a full account of queries in MongoDB.

In this procedure, you query for specific documents in the `testData` collection by passing a “query document” as a parameter to the `find()` method. A query document specifies the criteria the query must match to return a document.

In the mongo shell, query for all documents where the `x` field has a value of 18 by passing the `{ x : 18 }` query document as a parameter to the `find()` method:

```
db.testData.find( { x : 18 } )
```

MongoDB returns one document that fits this criteria:

```
{ "_id" : ObjectId("51a7dc7b2cacf40b79990bf7"), "x" : 18 }
```

Return a Single Document from a Collection

With the `findOne()` method you can return a single *document* from a MongoDB collection. The `findOne()` method takes the same parameters as `find()`, but returns a document rather than a cursor.

To retrieve one document from the `testData` collection, issue the following command:

```
db.testData.findOne()
```

For more information on querying for documents, see the [Query Documents](#) (page 57) and [Read Operations](#) (page 29) documentation.

Limit the Number of Documents in the Result Set

To increase performance, you can constrain the size of the result by limiting the amount of data your application must receive over the network.

To specify the maximum number of documents in the result set, call the `limit()` method on a cursor, as in the following command:

```
db.testData.find().limit(3)
```

MongoDB will return the following result, with different *ObjectId* (page 127) values:

```
{ "_id" : ObjectId("51a7dc7b2cacf40b79990be6"), "x" : 1 }
{ "_id" : ObjectId("51a7dc7b2cacf40b79990be7"), "x" : 2 }
{ "_id" : ObjectId("51a7dc7b2cacf40b79990be8"), "x" : 3 }
```

Next Steps with MongoDB

For more information on manipulating the documents in a database as you continue to learn MongoDB, consider the following resources:

- [MongoDB CRUD Operations](#) (page 25)
- [SQL to MongoDB Mapping Chart](#) (page 82)
- [MongoDB Drivers and Client Libraries](#) (page 92)

1.2.2 Generate Test Data

This tutorial describes how to quickly generate test data as you need to test basic MongoDB operations.

Insert Multiple Documents Using a For Loop

You can add documents to a new or existing collection by using a JavaScript `for` loop run from the mongo shell.

1. From the mongo shell, insert new documents into the `testData` collection using the following `for` loop. If the `testData` collection does not exist, MongoDB creates the collection implicitly.

```
for (var i = 1; i <= 25; i++) db.testData.insert( { x : i } )
```

2. Use `find()` to query the collection:

```
db.testData.find()
```

The mongo shell displays the first 20 documents in the collection. Your [ObjectId](#) (page 127) values will be different:

```
{ "_id" : ObjectId("51a7dc7b2cacf40b79990be6"), "x" : 1 }
{ "_id" : ObjectId("51a7dc7b2cacf40b79990be7"), "x" : 2 }
{ "_id" : ObjectId("51a7dc7b2cacf40b79990be8"), "x" : 3 }
{ "_id" : ObjectId("51a7dc7b2cacf40b79990be9"), "x" : 4 }
{ "_id" : ObjectId("51a7dc7b2cacf40b79990bea"), "x" : 5 }
{ "_id" : ObjectId("51a7dc7b2cacf40b79990beb"), "x" : 6 }
{ "_id" : ObjectId("51a7dc7b2cacf40b79990bec"), "x" : 7 }
{ "_id" : ObjectId("51a7dc7b2cacf40b79990bed"), "x" : 8 }
{ "_id" : ObjectId("51a7dc7b2cacf40b79990bee"), "x" : 9 }
{ "_id" : ObjectId("51a7dc7b2cacf40b79990bef"), "x" : 10 }
{ "_id" : ObjectId("51a7dc7b2cacf40b79990bf0"), "x" : 11 }
{ "_id" : ObjectId("51a7dc7b2cacf40b79990bf1"), "x" : 12 }
{ "_id" : ObjectId("51a7dc7b2cacf40b79990bf2"), "x" : 13 }
{ "_id" : ObjectId("51a7dc7b2cacf40b79990bf3"), "x" : 14 }
{ "_id" : ObjectId("51a7dc7b2cacf40b79990bf4"), "x" : 15 }
{ "_id" : ObjectId("51a7dc7b2cacf40b79990bf5"), "x" : 16 }
{ "_id" : ObjectId("51a7dc7b2cacf40b79990bf6"), "x" : 17 }
{ "_id" : ObjectId("51a7dc7b2cacf40b79990bf7"), "x" : 18 }
{ "_id" : ObjectId("51a7dc7b2cacf40b79990bf8"), "x" : 19 }
{ "_id" : ObjectId("51a7dc7b2cacf40b79990bf9"), "x" : 20 }
```

1. The `find()` returns a cursor. To iterate the cursor and return more documents use the `it` operation in the mongo shell. The mongo shell will exhaust the cursor, and return the following documents:

```
{ "_id" : ObjectId("51a7dce92cacf40b79990bfc"), "x" : 21 }
{ "_id" : ObjectId("51a7dce92cacf40b79990bfd"), "x" : 22 }
{ "_id" : ObjectId("51a7dce92cacf40b79990bfe"), "x" : 23 }
{ "_id" : ObjectId("51a7dce92cacf40b79990bff"), "x" : 24 }
{ "_id" : ObjectId("51a7dce92cacf40b79990c00"), "x" : 25 }
```

Insert Multiple Documents with a mongo Shell Function

You can create a JavaScript function in your shell session to generate the above data. The `insertData()` JavaScript function, shown here, creates new data for use in testing or training by either creating a new collection or appending data to an existing collection:

```
function insertData(dbName, colName, num) {

    var col = db.getSiblingDB(dbName).getCollection(colName);

    for (i = 0; i < num; i++) {
        col.insert({x:i});
    }

    print(col.count());
}
```

The `insertData()` function takes three parameters: a database, a new or existing collection, and the number of documents to create. The function creates documents with an `x` field that is set to an incremented integer, as in the following example documents:

```
{ "_id" : ObjectId("51a4da9b292904caffcff6eb"), "x" : 0 }
{ "_id" : ObjectId("51a4da9b292904caffcff6ec"), "x" : 1 }
{ "_id" : ObjectId("51a4da9b292904caffcff6ed"), "x" : 2 }
```

Store the function in your `.mongorc.js` file. The mongo shell loads the function for you every time you start a session.

Example

Specify database name, collection name, and the number of documents to insert as arguments to `insertData()`.

```
insertData("test", "testData", 400)
```

This operation inserts 400 documents into the `testData` collection in the `test` database. If the collection and database do not exist, MongoDB creates them implicitly before inserting documents.

See also:

[MongoDB CRUD Concepts](#) (page 27) and [Data Models](#) (page 95).

MongoDB CRUD Operations

MongoDB provides rich semantics for reading and manipulating data. CRUD stands for *create*, *read*, *update*, and *delete*. These terms are the foundation for all interactions with the database.

[MongoDB CRUD Introduction](#) (page 25) An introduction to the MongoDB data model as well as queries and data manipulations.

[MongoDB CRUD Concepts](#) (page 27) The core documentation of query and data manipulation.

[MongoDB CRUD Tutorials](#) (page 55) Examples of basic query and data modification operations.

[MongoDB CRUD Reference](#) (page 80) Reference material for the query and data manipulation interfaces.

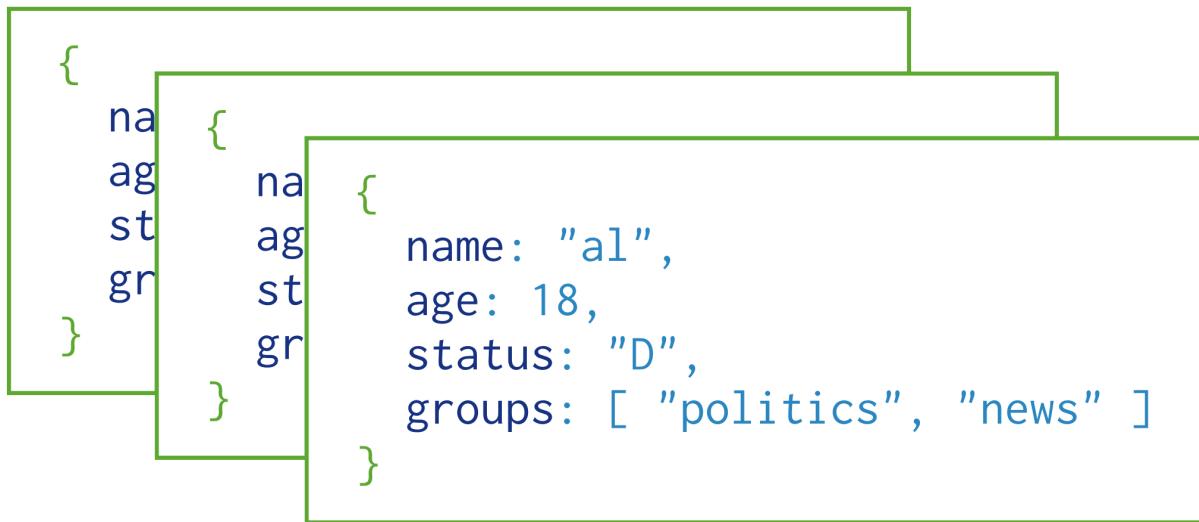
2.1 MongoDB CRUD Introduction

MongoDB stores data in the form of *documents*, which are JSON-like field and value pairs. Documents are analogous to structures in programming languages that associate keys with values, where keys may hold other pairs of keys and values (e.g. dictionaries, hashes, maps, and associative arrays). Formally, MongoDB documents are *BSON* documents, which is a binary representation of *JSON* with additional type information. For more information, see [Documents](#) (page 120).

```
{
  name: "sue",           ← field: value
  age: 26,               ← field: value
  status: "A",            ← field: value
  groups: [ "news", "sports" ] ← field: value
}
```

Figure 2.1: A MongoDB document.

MongoDB stores all documents in *collections*. A collection is a group of related documents that have a set of shared common indexes. Collections are analogous to a table in relational databases.



Collection

Figure 2.2: A collection of MongoDB documents.

2.1.1 Database Operations

Query

In MongoDB a query targets a specific collection of documents. Queries specify criteria, or conditions, that identify the documents that MongoDB returns to the clients. A query may include a *projection* that specifies the fields from the matching documents to return. You can optionally modify queries to impose limits, skips, and sort orders.

In the following diagram, the query process specifies a query criteria and a sort modifier:

Data Modification

Data modification refers to operations that create, update, or delete data. In MongoDB, these operations modify the data of a single *collection*. For the update and delete operations, you can specify the criteria to select the documents to update or remove.

In the following diagram, the insert operation adds a new document to the `users` collection.

2.1.2 Related Features

Indexes

To enhance the performance of common queries and updates, MongoDB has full support for secondary indexes. These indexes allow applications to store a *view* of a portion of the collection in an efficient data structure. Most indexes store an ordered representation of all values of a field or a group of fields. Indexes may also *enforce uniqueness* (page 330), store objects in a *geospatial representation* (page 322), and facilitate *text search* (page 328).

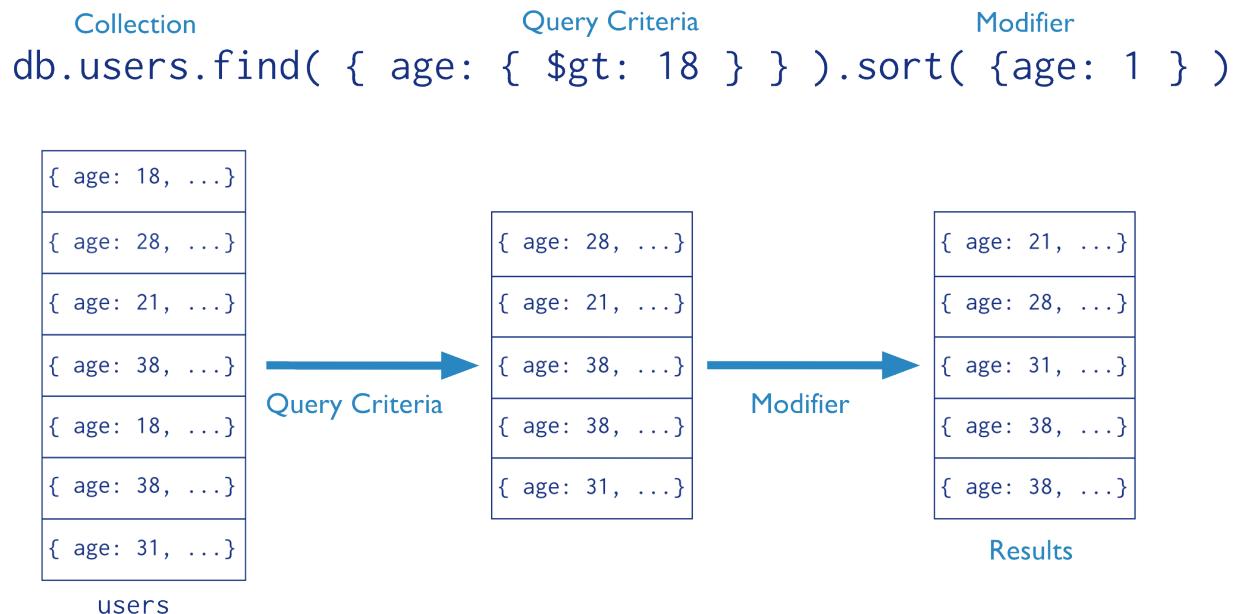


Figure 2.3: The stages of a MongoDB query with a query criteria and a sort modifier.

Read Preference

For replica sets and sharded clusters with replica set components, applications specify [read preferences](#) (page 401). A read preference determines how the client direct read operations to the set.

Write Concern

Applications can also control the behavior of write operations using [write concern](#) (page 44). Particularly useful for deployments with replica sets, the write concern semantics allow clients to specify the assurance that MongoDB provides when reporting on the success of a write operation.

Aggregation

In addition to the basic queries, MongoDB provides several data aggregation features. For example, MongoDB can return counts of the number of documents that match a query, or return the number of distinct values for a field, or process a collection of documents using a versatile stage-based data processing pipeline or map-reduce operations.

2.2 MongoDB CRUD Concepts

The [Read Operations](#) (page 29) and [Write Operations](#) (page 40) documents introduce the behavior and operations of read and write operations for MongoDB deployments.

Read Operations (page 29) Introduces all operations that select and return documents to clients, including the query specifications.

Cursors (page 33) Queries return iterable objects, called cursors, that hold the full result set of the query request.

Query Optimization (page 34) Analyze and improve query performance.

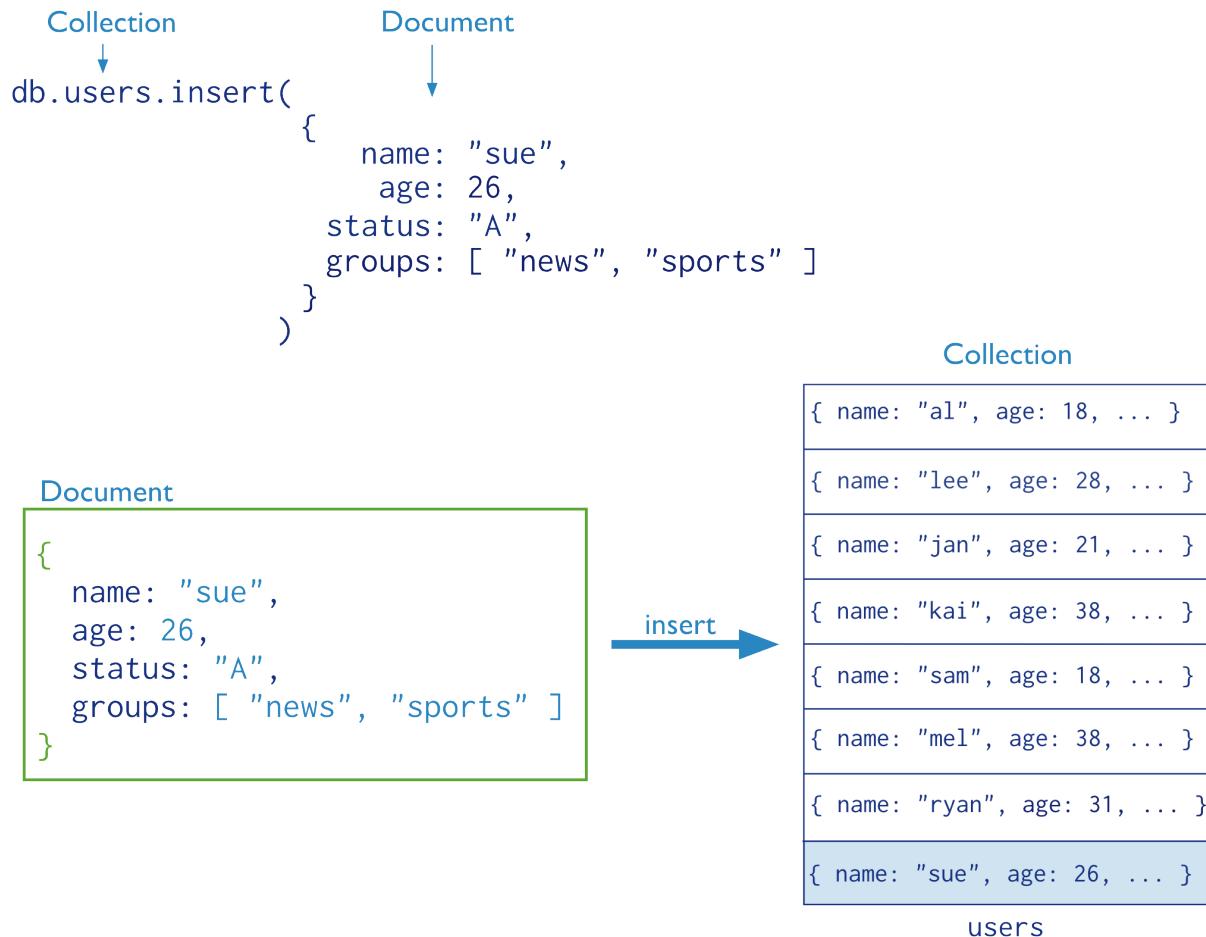


Figure 2.4: The stages of a MongoDB insert operation.

Distributed Queries (page 36) Describes how *sharded clusters* and *replica sets* affect the performance of read operations.

Write Operations (page 40) Introduces data create and modify operations, their behavior, and performances.

Write Concern (page 44) Describes the kind of guarantee MongoDB provides when reporting on the success of a write operation.

Distributed Write Operations (page 47) Describes how MongoDB directs write operations on *sharded clusters* and *replica sets* and the performance characteristics of these operations.

2.2.1 Read Operations

Read operations, or *queries*, retrieve data stored in the database. In MongoDB, queries select *documents* from a single *collection*.

Queries specify criteria, or conditions, that identify the documents that MongoDB returns to the clients. A query may include a *projection* that specifies the fields from the matching documents to return. The projection limits the amount of data that MongoDB returns to the client over the network.

Query Interface

For query operations, MongoDB provide a `db.collection.find()` method. The method accepts both the query criteria and projections and returns a *cursor* (page 33) to the matching documents. You can optionally modify the query to impose limits, skips, and sort orders.

The following diagram highlights the components of a MongoDB query operation:

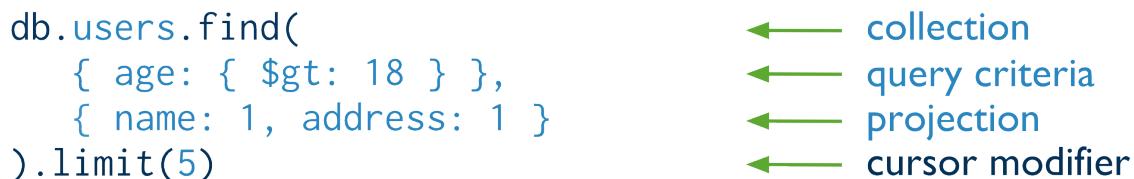


Figure 2.5: The components of a MongoDB find operation.

The next diagram shows the same query in SQL:



Figure 2.6: The components of a SQL SELECT statement.

Example

```
db.users.find( { age: { $gt: 18 } }, { name: 1, address: 1 } ).limit(5)
```

This query selects the documents in the `users` collection that match the condition `age` is greater than 18. To specify the greater than condition, query criteria uses the greater than (i.e. `$gt`) *query selection operator*. The query returns at most 5 matching documents (or more precisely, a cursor to those documents). The matching documents will return with only the `_id`, `name` and `address` fields. See [Projections](#) (page 30) for details.

See

[SQL to MongoDB Mapping Chart](#) (page 82) for additional examples of MongoDB queries and the corresponding SQL statements.

Query Behavior

MongoDB queries exhibit the following behavior:

- All queries in MongoDB address a *single* collection.
- You can modify the query to impose `limits`, `skips`, and `sort orders`.
- The order of documents returned by a query is not defined and is not necessarily consistent unless you specify a `sort()`.
- Operations that [modify existing documents](#) (page 64) (i.e. *updates*) use the same query syntax as queries to select documents to update.
- In [aggregation](#) (page 277) pipeline, the `$match` pipeline stage provides access to MongoDB queries.

MongoDB provides a `db.collection.findOne()` method as a special case of `find()` that returns a single document.

Query Statements

Consider the following diagram of the query process that specifies a query criteria and a sort modifier:

In the diagram, the query selects documents from the `users` collection. Using a *query selection operator* to define the conditions for matching documents, the query selects documents that have `age` greater than (i.e. `$gt`) 18. Then the `sort()` modifier sorts the results by `age` in ascending order.

For additional examples of queries, see [Query Documents](#) (page 57).

Projections

Queries in MongoDB return all fields in all matching documents by default. To limit the amount of data that MongoDB sends to applications, include a *projection* in the queries. By projecting results with a subset of fields, applications reduce their network overhead and processing requirements.

Projections, which are the the *second* argument to the `find()` method, may either specify a list of fields to return or list fields to exclude in the result documents.

Important: Except for excluding the `_id` field in inclusive projections, you cannot mix exclusive and inclusive projections.

Consider the following diagram of the query process that specifies a query criteria and a projection:

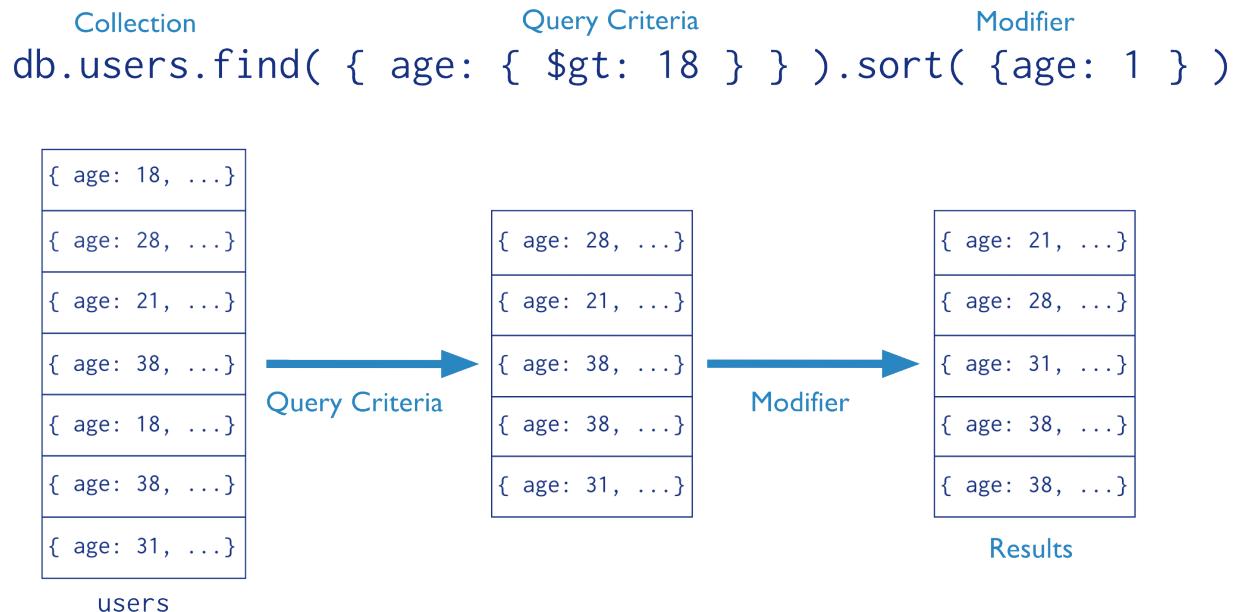


Figure 2.7: The stages of a MongoDB query with a query criteria and a sort modifier.

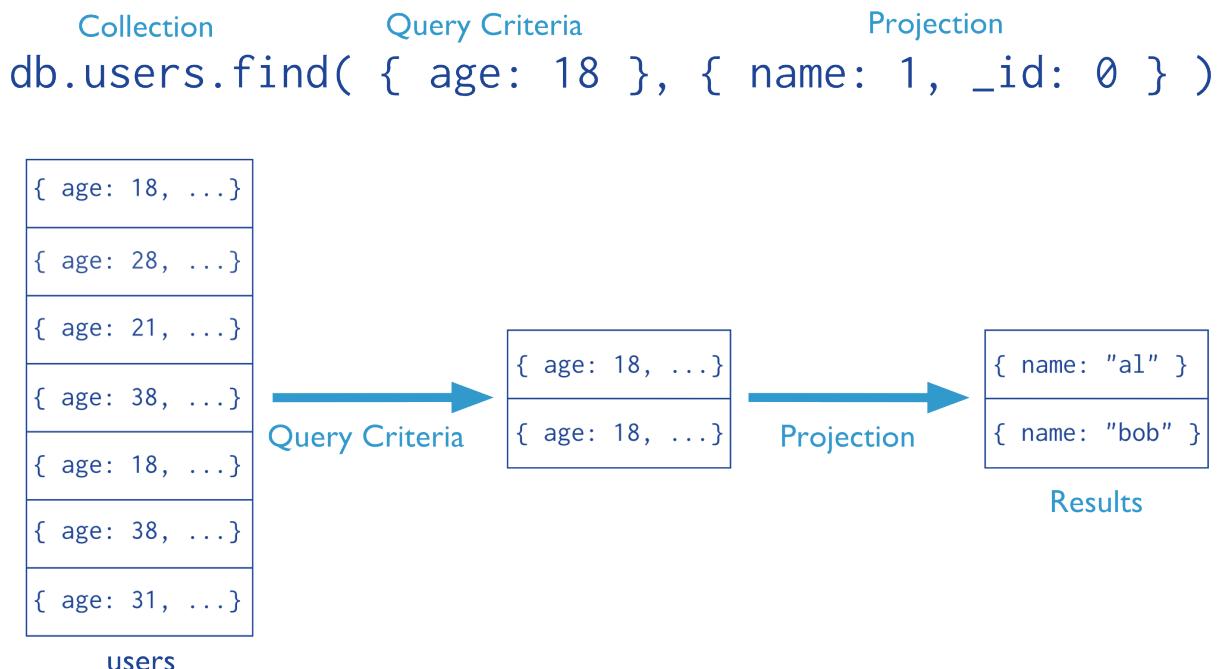


Figure 2.8: The stages of a MongoDB query with a query criteria and projection. MongoDB only transmits the projected data to the clients.

In the diagram, the query selects from the `users` collection. The criteria matches the documents that have `age` equal to 18. Then the projection specifies that only the `name` field should return in the matching documents.

Projection Examples

Exclude One Field From a Result Set

```
db.records.find( { "user_id": { $lt: 42} }, { history: 0 } )
```

This query selects a number of documents in the `records` collection that match the query `{ "user_id": { $lt: 42} }`, but excludes the `history` field.

Return Two fields and the `_id` Field

```
db.records.find( { "user_id": { $lt: 42} }, { "name": 1, "email": 1 } )
```

This query selects a number of documents in the `records` collection that match the query `{ "user_id": { $lt: 42} }`, but returns documents that have the `_id` field (implicitly included) as well as the `name` and `email` fields.

Return Two Fields and Exclude `_id`

```
db.records.find( { "user_id": { $lt: 42} }, { "_id": 0, "name": 1 , "email": 1 } )
```

This query selects a number of documents in the `records` collection that match the query `{ "user_id": { $lt: 42} }`, but only returns the `name` and `email` fields.

See

[Limit Fields to Return from a Query](#) (page 61) for more examples of queries with projection statements.

Projection Behavior

MongoDB projections have the following properties:

- In MongoDB, the `_id` field is always included in results unless explicitly excluded.
- For fields that contain arrays, MongoDB provides the following projection operators: `$elemMatch`, `$slice`, `$`.
- For related projection functionality in the [aggregation framework](#) (page 277) pipeline, use the `$project` pipeline stage.

Related Concepts

The following documents further describe read operations:

[Cursors](#) (page 33) Queries return iterable objects, called cursors, that hold the full result set of the query request.

[Query Optimization](#) (page 34) Analyze and improve query performance.

[Query Plans](#) (page 35) MongoDB processes and executes using plans developed to return results as efficiently as possible.

[Distributed Queries](#) (page 36) Describes how *sharded clusters* and *replica sets* affect the performance of read operations.

Cursors

In the mongo shell, the primary method for the read operation is the `db.collection.find()` method. This method queries a collection and returns a *cursor* to the returning documents.

To access the documents, you need to iterate the cursor. 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 times¹ to print up to the first 20 documents in the results.

For example, in the mongo shell, the following read operation queries the `inventory` collection for documents that have `type` equal to '`food`' and automatically print up to the first 20 matching documents:

```
db.inventory.find( { type: 'food' } );
```

To manually iterate the cursor to access the documents, see [Iterate a Cursor in the mongo Shell](#) (page 62).

Cursor Behaviors

Closure of Inactive Cursors By default, the server will automatically close the cursor after 10 minutes of inactivity or if client has exhausted the cursor. To override this behavior, you can specify the `noTimeout` wire protocol flag² in your query; however, you should either close the cursor manually or exhaust the cursor. In the mongo shell, you can set the `noTimeout` flag:

```
var myCursor = db.inventory.find().addOption(DBQuery.Option.noTimeout);
```

See your [driver](#) (page 92) documentation for information on setting the `noTimeout` flag. For the mongo shell, see `cursor.addOption()` for a complete list of available cursor flags.

Cursor Isolation Because the cursor is not isolated during its lifetime, intervening write operations on a document may result in a cursor that returns a document more than once if that document has changed. To handle this situation, see the information on [snapshot mode](#) (page 582).

Cursor Batches The MongoDB server returns the query results in batches. Batch size will not exceed the *maximum BSON document size*. For most queries, the *first* batch returns 101 documents or just enough documents to exceed 1 megabyte. Subsequent batch size is 4 megabytes. To override the default size of the batch, see `batchSize()` and `limit()`.

For queries that include a sort operation *without* an index, the server must load all the documents in memory to perform the sort and will return all documents in the first batch.

As you iterate through the cursor and reach the end of the returned batch, if there are more results, `cursor.next()` will perform a `getmore` operation to retrieve the next batch. To see how many documents remain in the batch as you iterate the cursor, you can use the `objsLeftInBatch()` method, as in the following example:

```
var myCursor = db.inventory.find();
var myFirstDocument = myCursor.hasNext() ? myCursor.next() : null;
myCursor objsLeftInBatch();
```

¹ You can use the `DBQuery.shellBatchSize` to change the number of iteration from the default value 20. See [Executing Queries](#) (page 209) for more information.

²<http://docs.mongodb.org/meta-driver/latest/legacy/mongodb-wire-protocol>

Cursor Information You can use the command `cursorInfo` to retrieve the following information on cursors:

- total number of open cursors
- size of the client cursors in current use
- number of timed out cursors since the last server restart

Consider the following example:

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

The result from the command returns the following document:

```
{  
  "totalOpen" : <number>,  
  "clientCursors_size" : <number>,  
  "timedOut" : <number>,  
  "ok" : 1  
}
```

Query Optimization

Indexes improve the efficiency of read operations by reducing the amount of data that query operations need to process. This simplifies the work associated with fulfilling queries within MongoDB.

Create an Index to Support Read Operations If your application queries a collection on a particular field or fields, then an index on the queried field or fields can prevent the query from scanning the whole collection to find and return the query results. For more information about indexes, see the [complete documentation of indexes in MongoDB](#) (page 314).

Example

An application queries the `inventory` collection on the `type` field. The value of the `type` field is user-driven.

```
var typeValue = <someUserInput>;  
db.inventory.find( { type: typeValue } );
```

To improve the performance of this query, add an ascending, or a descending, index to the `inventory` collection on the `type` field.³ In the `mongo` shell, you can create indexes using the `db.collection.ensureIndex()` method:

```
db.inventory.ensureIndex( { type: 1 } )
```

This index can prevent the above query on `type` from scanning the whole collection to return the results.

To analyze the performance of the query with an index, see [Analyze Query Performance](#) (page 63).

In addition to optimizing read operations, indexes can support sort operations and allow for a more efficient storage utilization. See `db.collection.ensureIndex()` and [Indexing Tutorials](#) (page 334) for more information about index creation.

³ For single-field indexes, the selection between ascending and descending order is immaterial. For compound indexes, the selection is important. See [indexing order](#) (page 319) for more details.

Query Selectivity Some query operations are not selective. These operations cannot use indexes effectively or cannot use indexes at all.

The inequality operators `$nin` and `$ne` are not very selective, as they often match a large portion of the index. As a result, in most cases, a `$nin` or `$ne` query with an index may perform no better than a `$nin` or `$ne` query that must scan all documents in a collection.

Queries that specify regular expressions, with inline JavaScript regular expressions or `$regex` operator expressions, cannot use an index with one exception. Queries that specify regular expression *with anchors* at the beginning of a string *can* use an index.

Covering a Query An index *covers* (page 365) a query, a *covered query*, when:

- all the fields in the *query* (page 57) are part of that index, **and**
- all the fields returned in the documents that match the query are in the same index.

For these queries, MongoDB does not need to inspect documents outside of the index. This is often more efficient than inspecting entire documents.

Example

Given a collection `inventory` with the following index on the `type` and `item` fields:

```
{ type: 1, item: 1 }
```

This index will cover the following query on the `type` and `item` fields, which returns only the `item` field:

```
db.inventory.find( { type: "food", item:/^c/ },
    { item: 1, _id: 0 } )
```

However, the index will **not** cover the following query, which returns the `item` field **and** the `_id` field:

```
db.inventory.find( { type: "food", item:/^c/ },
    { item: 1 } )
```

See *Create Indexes that Support Covered Queries* (page 365) for more information on the behavior and use of covered queries.

Query Plans

The MongoDB query optimizer processes queries and chooses the most efficient query plan for a query given the available indexes. The query system then uses this query plan each time the query runs. The query optimizer occasionally reevaluates query plans as the content of the collection changes to ensure optimal query plans.

You can use the `explain()` method to view statistics about the query plan for a given query. This information can help as you develop *indexing strategies* (page 363).

Query Optimization To create a new query plan, the query optimizer:

1. runs the query against several candidate indexes in parallel.
2. records the matches in a common results buffer or buffers.
 - If the candidate plans include only *ordered query plans*, there is a single common results buffer.
 - If the candidate plans include only *unordered query plans*, there is a single common results buffer.

- If the candidate plans include *both ordered query plans* and *unordered query plans*, there are two common results buffers, one for the ordered plans and the other for the unordered plans.

If an index returns a result already returned by another index, the optimizer skips the duplicate match. In the case of the two buffers, both buffers are de-duped.

3. stops the testing of candidate plans and selects an index when one of the following events occur:

- An *unordered query plan* has returned all the matching results; *or*
- An *ordered query plan* has returned all the matching results; *or*
- An *ordered query plan* has returned a threshold number of matching results:
 - Version 2.0: Threshold is the query batch size. The default batch size is 101.
 - Version 2.2: Threshold is 101.

The selected index becomes the index specified in the query plan; future iterations of this query or queries with the same query pattern will use this index. Query pattern refers to query select conditions that differ only in the values, as in the following two queries with the same query pattern:

```
db.inventory.find( { type: 'food' } )
db.inventory.find( { type: 'utensil' } )
```

Query Plan Revision As collections change over time, the query optimizer deletes the query plan and re-evaluates after any of the following events:

- The collection receives 1,000 write operations.
- The `reIndex` rebuilds the index.
- You add or drop an index.
- The `mongod` process restarts.

Distributed Queries

Read Operations to Sharded Clusters *Sharded clusters* allow you to partition a data set among a cluster of `mongod` instances in a way that is nearly transparent to the application. For an overview of sharded clusters, see the [Sharding](#) (page 487) section of this manual.

For a sharded cluster, applications issue operations to one of the `mongos` instances associated with the cluster.

Read operations on sharded clusters are most efficient when directed to a specific shard. Queries to sharded collections should include the collection's [shard key](#) (page 499). When a query includes a shard key, the `mongos` can use cluster metadata from the [config database](#) (page 496) to route the queries to shards.

If a query does not include the shard key, the `mongos` must direct the query to *all* shards in the cluster. These *scatter gather* queries can be inefficient. On larger clusters, scatter gather queries are unfeasible for routine operations.

For more information on read operations in sharded clusters, see the [Sharded Cluster Query Routing](#) (page 503) and [Shard Keys](#) (page 499) sections.

Read Operations to Replica Sets *Replica sets* use [read preferences](#) to determine where and how to route read operations to members of the replica set. By default, MongoDB always reads data from a replica set's *primary*. You can modify that behavior by changing the [read preference mode](#) (page 483).

You can configure the [read preference mode](#) (page 483) on a per-connection or per-operation basis to allow reads from *secondaries* to:

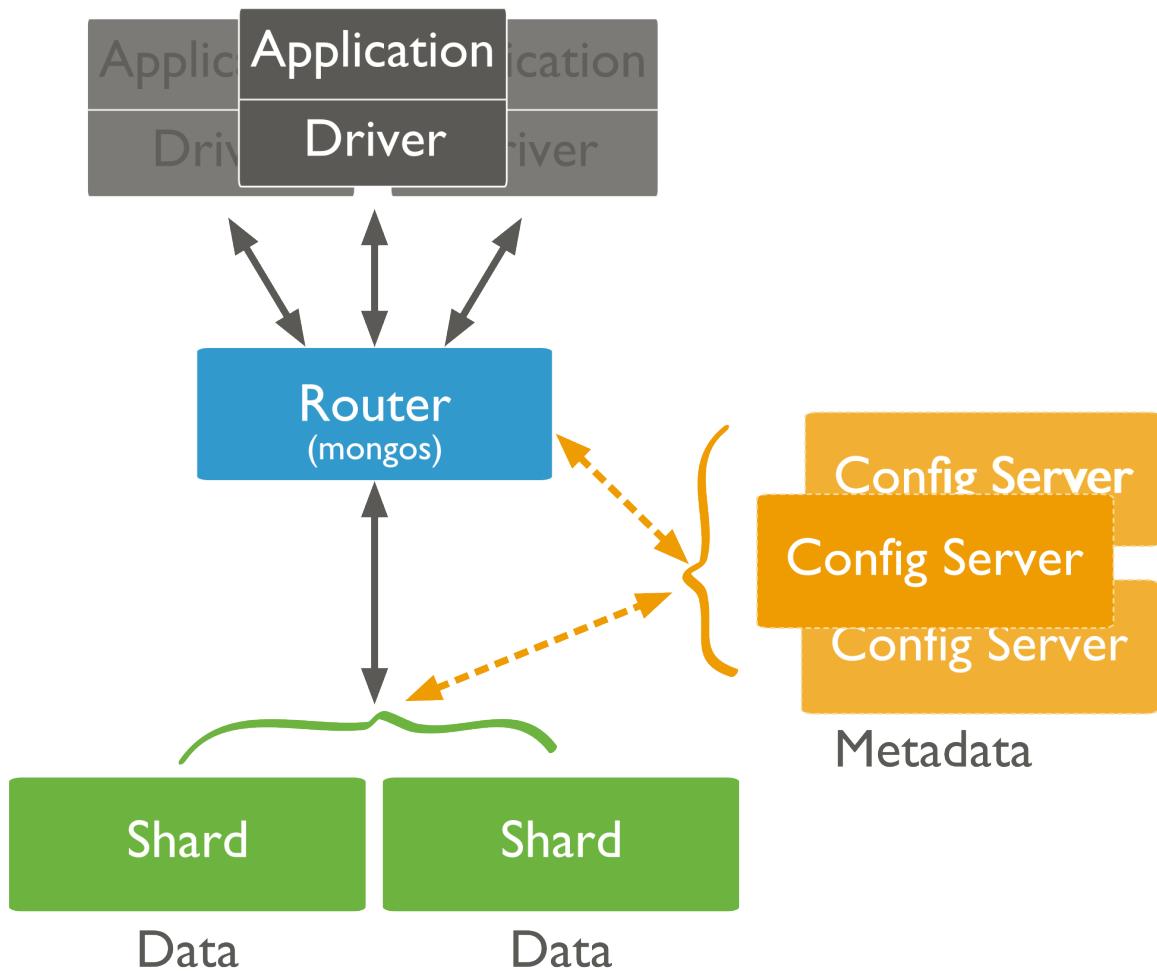


Figure 2.9: Diagram of a sharded cluster.

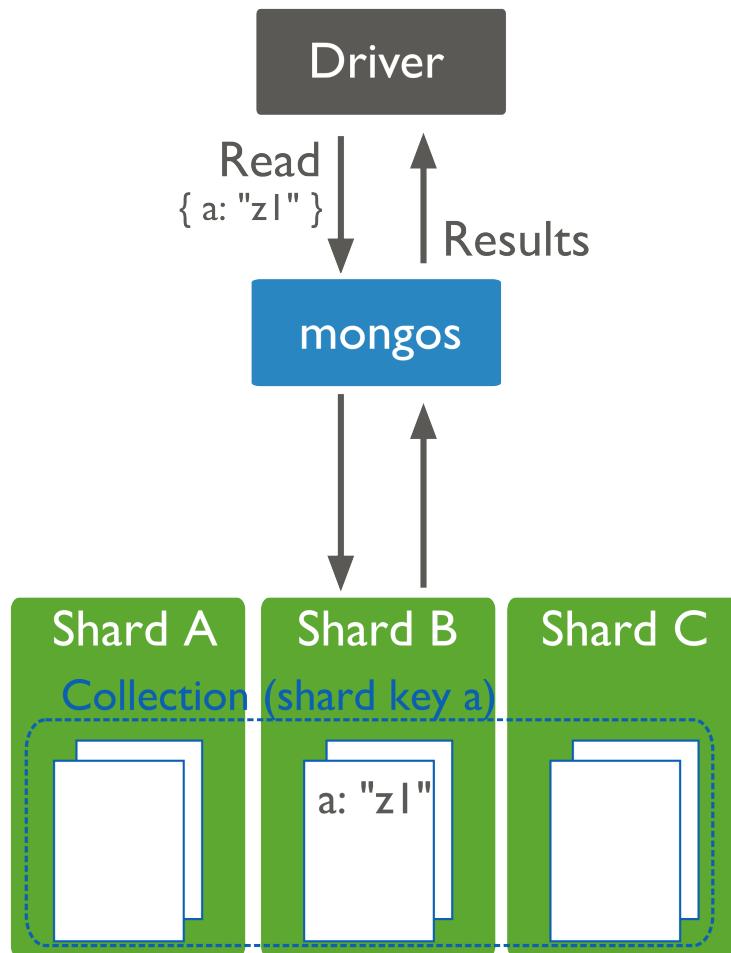


Figure 2.10: Read operations to a sharded cluster. Query criteria includes the shard key. The query router mongos can target the query to the appropriate shard or shards.

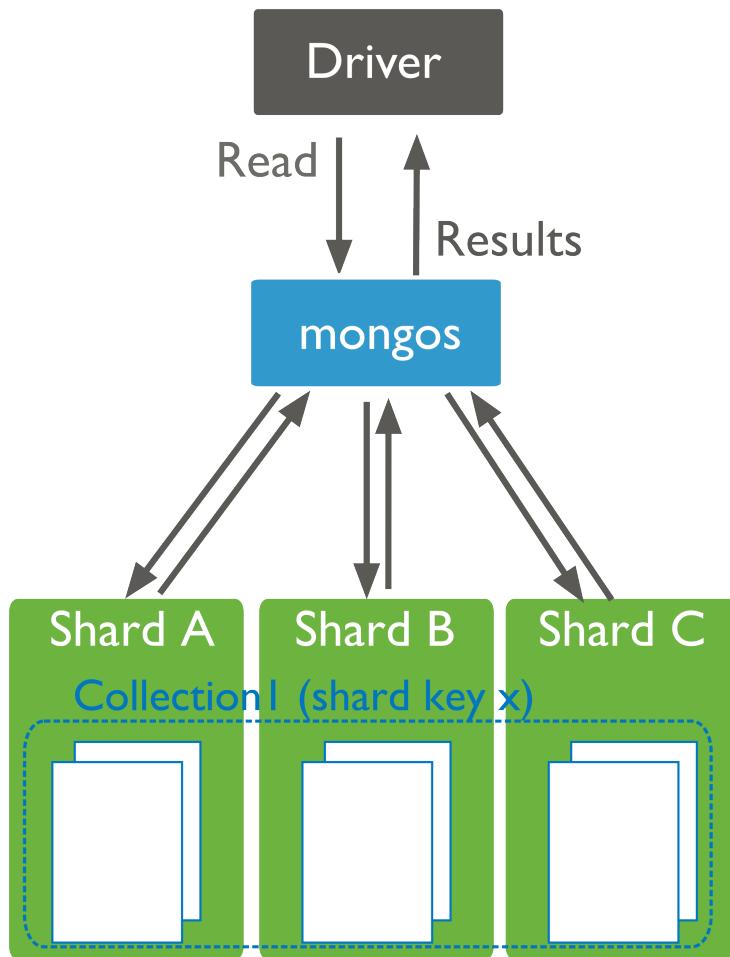


Figure 2.11: Read operations to a sharded cluster. Query criteria does not include the shard key. The query router mongos must broadcast query to all shards for the collection.

- reduce latency in multi-data-center deployments,
- improve read throughput by distributing high read-volumes (relative to write volume),
- for backup operations, and/or
- to allow reads during [failover](#) (page 392) situations.

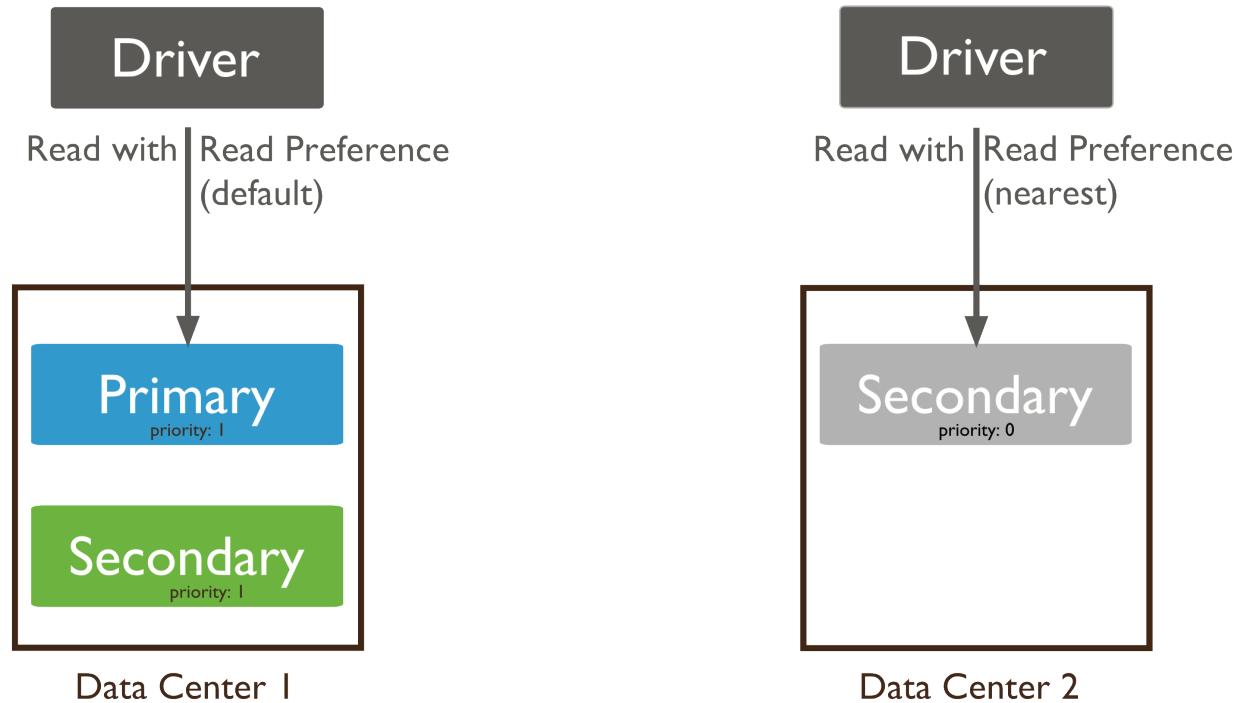


Figure 2.12: Read operations to a replica set. Default read preference routes the read to the primary. Read preference of nearest routes the read to the nearest member.

Read operations from secondary members of replica sets are not guaranteed to reflect the current state of the primary, and the state of secondaries will trail the primary by some amount of time. Often, applications don't rely on this kind of strict consistency, but application developers should always consider the needs of their application before setting read preference.

For more information on read preference or on the read preference modes, see [Read Preference](#) (page 401) and [Read Preference Modes](#) (page 483).

2.2.2 Write Operations

A write operation is any operation that creates or modifies data in the MongoDB instance. In MongoDB, write operations target a single *collection*. All write operations in MongoDB are atomic on the level of a single *document*.

There are three classes of write operations in MongoDB: insert, update, and remove. Insert operations add new data to a collection. Update operations modify an existing data, and remove operations delete data from a collection. No insert, update, or remove can affect more than one document atomically.

For the update and remove operations, you can specify criteria, or conditions, that identify the documents to update or remove. These operations use the same query syntax to specify the criteria as [read operations](#) (page 29).

After issuing these modification operations, MongoDB allows applications to determine the level of acknowledgment returned from the database. See [Write Concern](#) (page 44).

Create

Create operations add new *documents* to a collection. In MongoDB, the `db.collection.insert()` method performs create operations.

The following diagram highlights the components of a MongoDB insert operation:

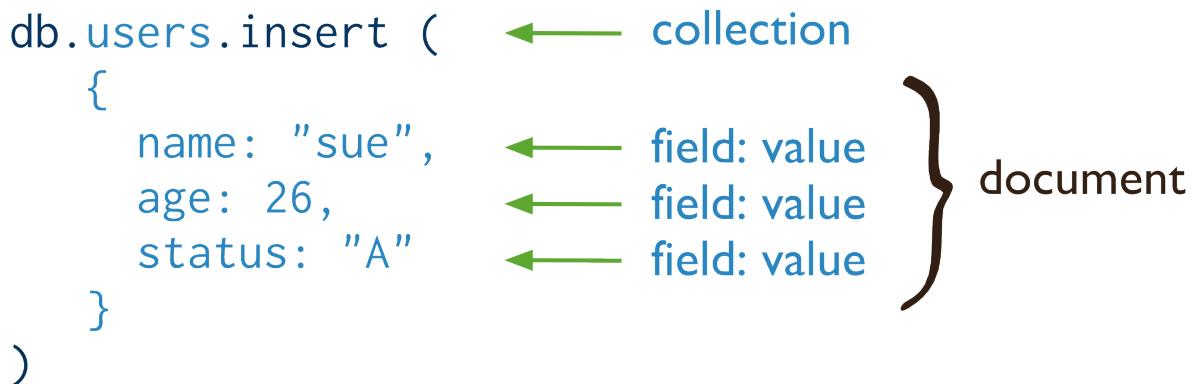


Figure 2.13: The components of a MongoDB insert operations.

The following diagram shows the same query in SQL:

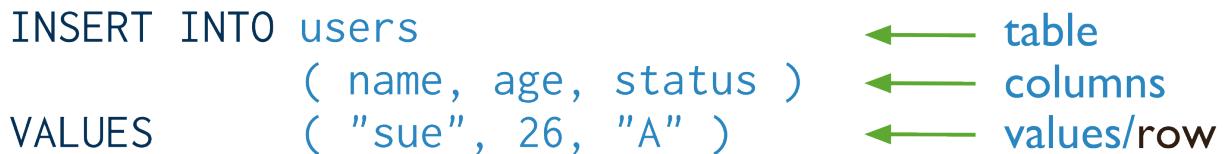


Figure 2.14: The components of a SQL INSERT statement.

Example

```
db.users.insert (
  {
    name: "sue",
    age: 26,
    status: "A"
  }
)
```

This operation inserts a new documents into the `users` collection. The new document has four fields `name`, `age`, and `status`, and an `_id` field. MongoDB always adds the `_id` field to the new document if the field does not exist.

For more information, see `db.collection.insert()` and [Insert Documents](#) (page 56).

An `upsert` is an operation that performs either an update of an existing document or an insert of a new document if the document to modify does not exist. With an `upsert`, applications do not need to make two separate calls to the database in order to decide between performing an update or an insert operation. Both the `db.collection.update()` method and the `db.collection.save()` method can perform an `upsert`. See

`db.collection.update()` and `db.collection.save()` for details on performing an upsert with these methods.

See

[SQL to MongoDB Mapping Chart](#) (page 82) for additional examples of MongoDB write operations and the corresponding SQL statements.

Insert Behavior

If you add a new document *without* the `_id` field, the client library or the `mongod` instance adds an `_id` field and populates the field with a unique *ObjectId*.

If you specify the `_id` field, the value must be unique within the collection. For operations with [write concern](#) (page 44), if you try to create a document with a duplicate `_id` value, `mongod` returns a duplicate key exception.

Update

Update operations modify existing *documents* in a *collection*. In MongoDB, `db.collection.update()` and the `db.collection.save()` methods perform update operations. The `db.collection.update()` method can accept a query criteria to determine which documents to update as well as an option to update multiple rows. The method can also accept options that affect its behavior such as the `multi` option to update multiple documents.

The following diagram highlights the components of a MongoDB update operation:

```
db.users.update(  
    { age: { $gt: 18 } },           ← collection  
    { $set: { status: "A" } },      ← update criteria  
    { multi: true }               ← update action  
)  
                                ← update option
```

Figure 2.15: The components of a MongoDB update operation.

The following diagram shows the same query in SQL:

```
UPDATE users           ← table  
SET     status = 'A'   ← update action  
WHERE   age > 18       ← update criteria
```

Figure 2.16: The components of a SQL UPDATE statement.

Example

```
db.users.update(
  { age: { $gt: 18 } },
  { $set: { status: "A" } },
  { multi: true }
)
```

This update operation on the `users` collection sets the `status` field to `A` for the documents that match the criteria of age greater than 18.

For more information, see `db.collection.update()` and `db.collection.save()`, and [Modify Documents](#) (page 64) for examples.

Update Behavior

By default, the `db.collection.update()` method updates a **single** document. However, with the `multi` option, `update()` can update all documents in a collection that match a query.

The `db.collection.update()` method either updates specific fields in the existing document or replaces the document. See `db.collection.update()` for details.

When performing update operations that increase the document size beyond the allocated space for that document, the update operation relocates the document on disk and may reorder the document fields depending on the type of update.

The `db.collection.save()` method replaces a document and can only update a single document. See `db.collection.save()` and [Insert Documents](#) (page 56) for more information

Delete

Delete operations remove documents from a collection. In MongoDB, `db.collection.remove()` method performs delete operations. The `db.collection.remove()` method can accept a query criteria to determine which documents to remove.

The following diagram highlights the components of a MongoDB remove operation:

```
db.users.remove(
  { status: "D" }
)
```

← collection
← remove criteria

Figure 2.17: The components of a MongoDB remove operation.

The following diagram shows the same query in SQL:

```
DELETE FROM users
WHERE status = 'D'
```

← table
← delete criteria

Figure 2.18: The components of a SQL DELETE statement.

Example

```
db.users.remove(  
  { status: "D" }  
)
```

This delete operation on the `users` collection removes all documents that match the criteria of `status` equal to D.

For more information, see `db.collection.remove()` method and [Remove Documents](#) (page 65).

Remove Behavior

By default, `db.collection.remove()` method removes all documents that match its query. However, the method can accept a flag to limit the delete operation to a single document.

Isolation of Write Operations

The modification of a single document is always atomic, even if the write operation modifies multiple sub-documents *within* that document. For write operations that modify multiple documents, the operation as a whole is not atomic, and other operations may interleave.

No other operations are atomic. You can, however, attempt to isolate a write operation that affects multiple documents using the `isolation` operator.

To isolate a sequence of write operations from other read and write operations, see [Perform Two Phase Commits](#) (page 66).

Related Concepts

The following documents further describe write operations:

[Write Concern](#) (page 44) Describes the kind of guarantee MongoDB provides when reporting on the success of a write operation.

[Distributed Write Operations](#) (page 47) Describes how MongoDB directs write operations on *sharded clusters* and *replica sets* and the performance characteristics of these operations.

[Write Operation Performance](#) (page 52) Introduces the performance constraints and factors for writing data to MongoDB deployments.

[Bulk Inserts in MongoDB](#) (page 53) Describe behaviors associated with inserting an array of documents.

[Record Padding](#) (page 54) When storing documents on disk, MongoDB reserves space to allow documents to grow efficiently during subsequent updates.

Write Concern

Write concern describes the guarantee that MongoDB provides when reporting on the success of a write operation. The strength of the write concerns determine the level of guarantee. When inserts, updates and deletes have a *weak* write concern, write operations return quickly. In some failure cases, write operations issued with weak write concerns may not persist. With *stronger* write concerns, clients wait after sending a write operation for MongoDB to confirm the write operations.

MongoDB provides different levels of write concern to better address the specific needs of applications. Clients may adjust write concern to ensure that the most important operations persist successfully to an entire MongoDB deployment. For other less critical operations, clients can adjust the write concern to ensure faster performance rather than ensure persistence to the entire deployment.

See also:

[Write Concern Reference](#) (page 80) for a reference of specific write concern configuration. Also consider [Write Operations](#) (page 40) for a general overview of write operations with MongoDB and [Write Concern for Replica Sets](#) (page 398) for considerations specific to replica sets.

Note: The [driver write concern](#) (page 654) change created a new connection class in all of the MongoDB drivers. The new class, called MongoClient, changes the default write concern. See the [release notes](#) (page 654) for this change and the release notes for your driver.

Write Concern Levels Clients issue write operations with some level of *write concern*. MongoDB has the following levels of conceptual write concern, listed from weakest to strongest:

Errors Ignored With an *errors ignored* write concern, MongoDB does not acknowledge write operations. With this level of write concern, the client cannot detect failed write operations. These errors include connection errors and mongod exceptions such as duplicate key exceptions for [unique indexes](#) (page 330). Although the *errors ignored* write concern provides fast performance, this performance gain comes at the cost of significant risks for data persistence and durability.

To set *errors ignored* write concern, specify w values of -1 to your driver.

Warning: Do not use *errors ignored* write concern in normal operation.

Unacknowledged With an *unacknowledged* write concern, MongoDB does not acknowledge the receipt of write operation. *Unacknowledged* is similar to *errors ignored*; however, drivers attempt to receive and handle network errors when possible. The driver's ability to detect network errors depends on the system's networking configuration.

To set *unacknowledged* write concern, specify w values of 0 to your driver.

Before the releases outlined in [Default Write Concern Change](#) (page 654), this was the default write concern.

Acknowledged With a receipt *acknowledged* write concern, the mongod confirms the receipt of the write operation. *Acknowledged* write concern allows clients to catch network, duplicate key, and other errors.

To set *acknowledged* write concern, specify w values of 1 to your driver.

MongoDB uses *acknowledged* write concern by default, after the releases outlined in [Default Write Concern Change](#) (page 654).

Internally, the default write concern calls `getLastError` with no arguments. For replica sets, you can define the default write concern settings in the [getLastErrorDefaults](#) (page 477). When `getLastErrorDefaults` (page 477) does not define a default write concern setting, `getLastError` defaults to basic receipt acknowledgment.

Journalized With a *journalized* write concern, the mongod confirms the write operation only after committing to the *journal*. A confirmed journal commit ensures *durability*, which guarantees that a write operation will survive a mongod shutdown. However, there is a window between journal commits when the write operation is not fully durable. See `journalCommitInterval` for more information on this window.

To set *journalized* write concern, specify w values of 1 and set the `j` option to true to your driver.

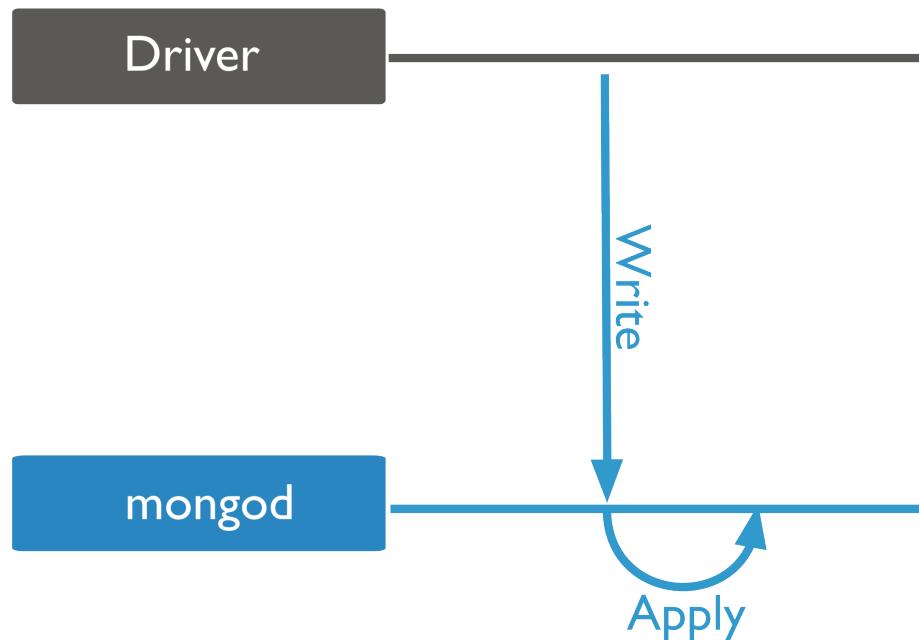


Figure 2.19: Write operation to a mongod instance with write concern of unacknowledged. The client does not wait for any acknowledgment.

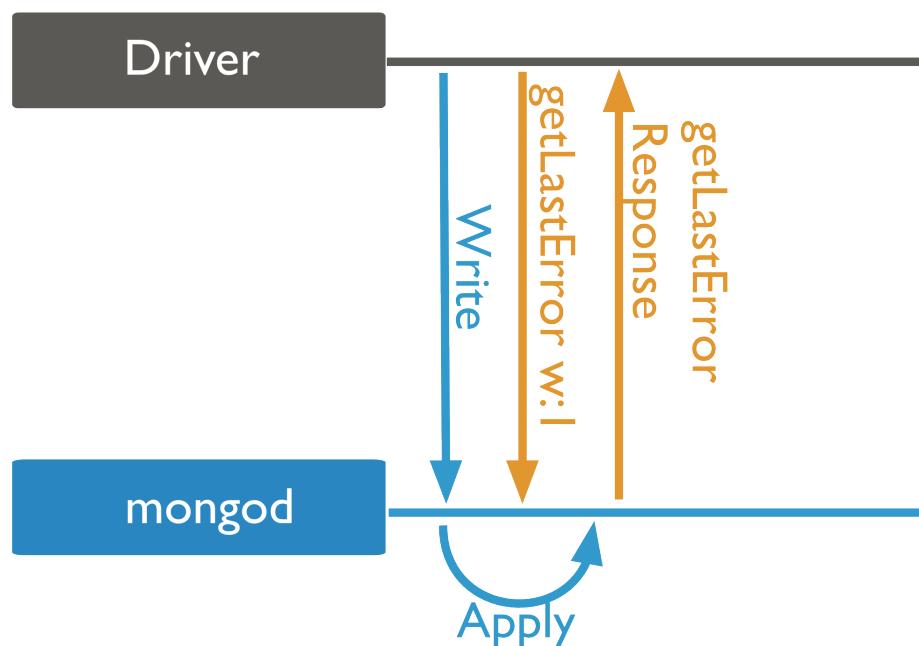


Figure 2.20: Write operation to a mongod instance with write concern of acknowledged. The client waits for acknowledgment of success or exception.

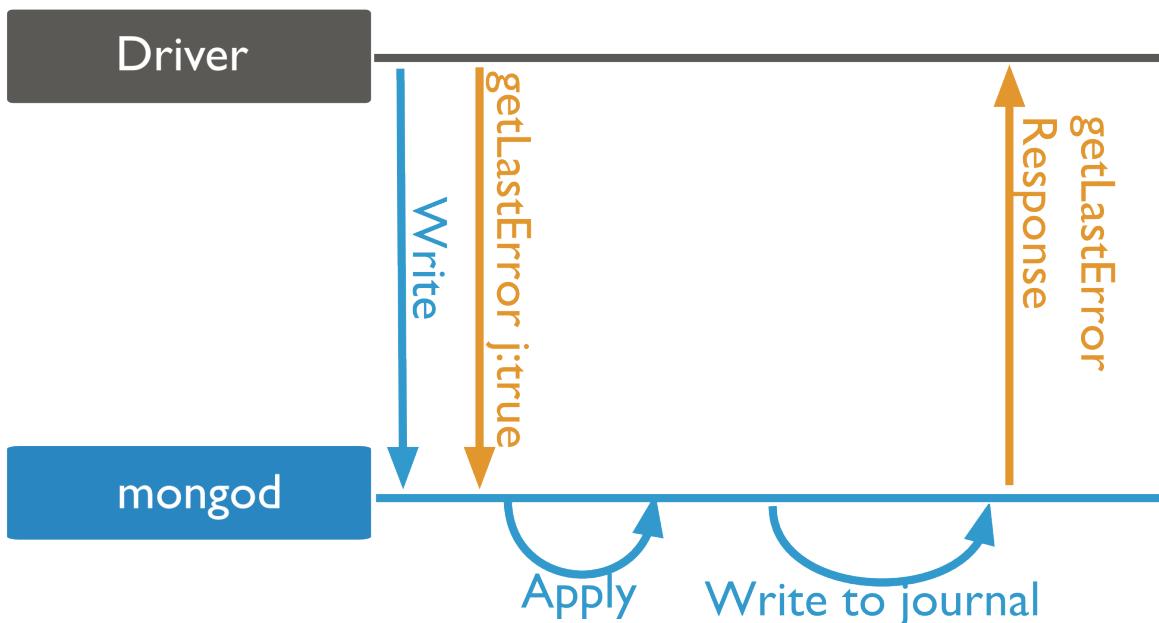


Figure 2.21: Write operation to a mongod instance with write concern of `journaled`. The mongod sends acknowledgement after it commits the write operation to the journal.

Receipt *acknowledged* without *journaled* provides the basis for write concern. Require *journaled* as part of the write concern to provide this durability guarantee. The mongod must have journaling enabled for the *journaled* write concern to have effect.

Note: Requiring *journaled* write concern in a replica set only requires a journal commit of the write operation to the primary of the set regardless of the level of *replica acknowledged* write concern.

Replica Acknowledged *Replica sets* add several considerations for write concern. Basic write concerns affect write operations on only one mongod instance. The `w` argument to `getLastError` provides *replica acknowledged* write concerns. With *replica acknowledged* you can guarantee that the write operation propagates to the members of a replica set. See [Write Concern Reference](#) (page 80) document for the values for `w` and [Write Concern for Replica Sets](#) (page 398) for more information.

To set *replica acknowledged* write concern, specify `w` values greater than 1 to your driver.

Note: Requiring *journaled* write concern in a replica set only requires a journal commit of the write operation to the primary of the set regardless of the level of *replica acknowledged* write concern.

Distributed Write Operations

Write Operations on Sharded Clusters For sharded collections in a *sharded cluster*, the mongos directs write operations from applications to the shards that are responsible for the specific *portion* of the data set. The mongos uses the cluster metadata from the [config database](#) (page 496) to route the write operation to the appropriate shards.

MongoDB partitions data in a sharded collection into *ranges* based on the values of the *shard key*. Then, MongoDB distributes these chunks to shards. The shard key determines the distribution of chunks to shards. This can affect the performance of write operations in the cluster.

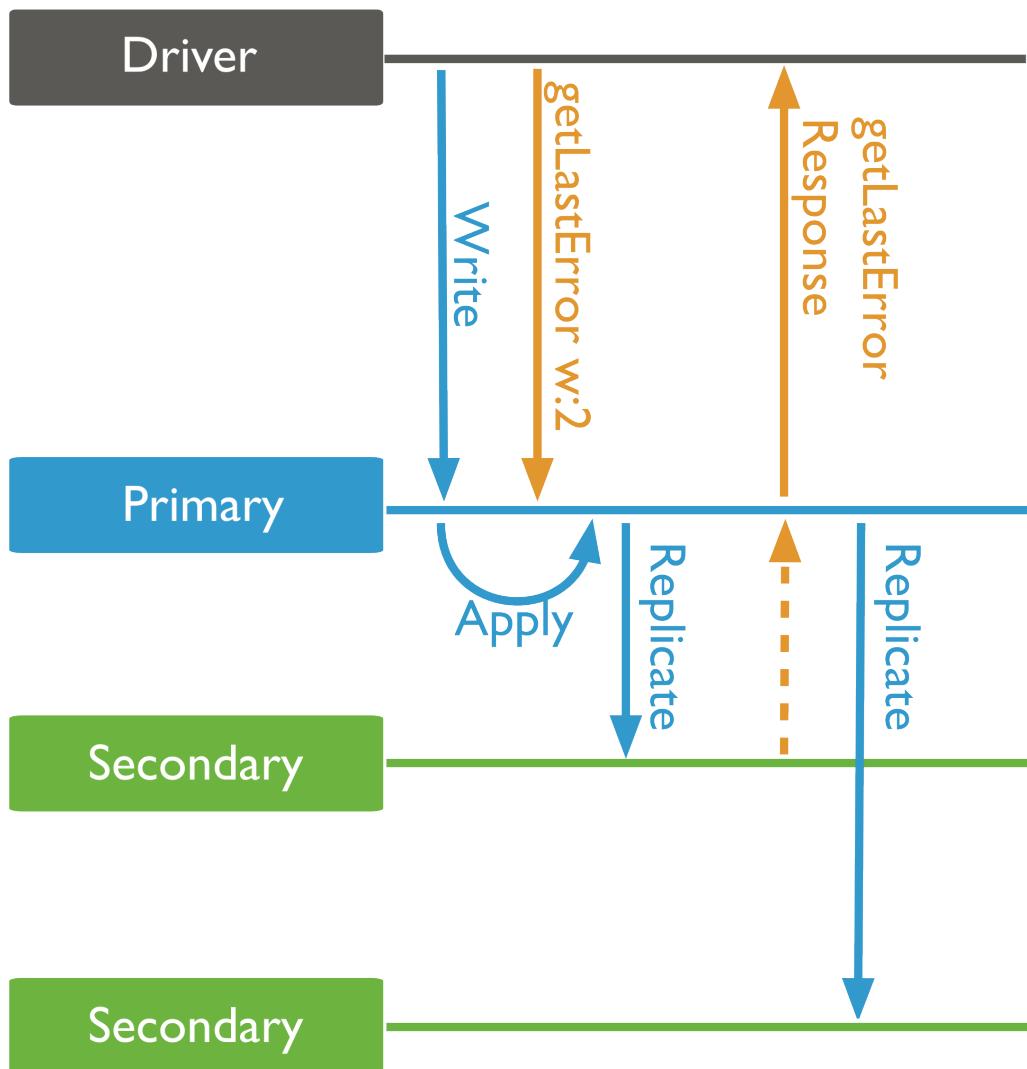


Figure 2.22: Write operation to a replica set with write concern level of `w:2` or write to the primary and at least one secondary.

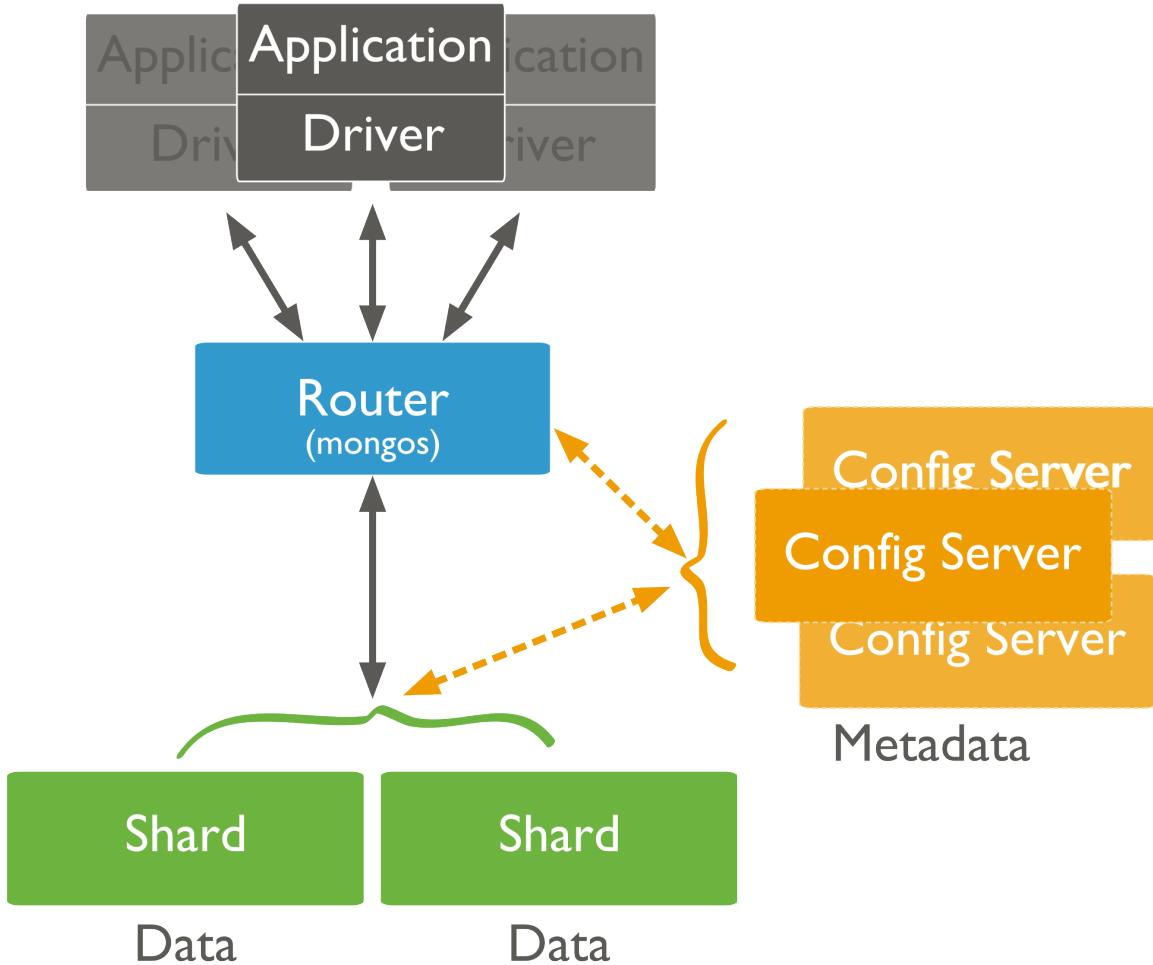


Figure 2.23: Diagram of a sharded cluster.

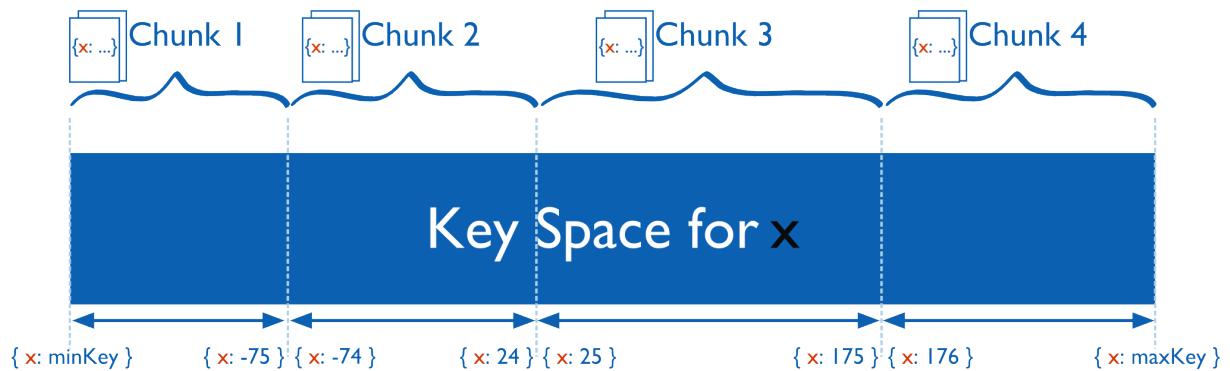


Figure 2.24: Diagram of the shard key value space segmented into smaller ranges or chunks.

Important: Update operations that affect a *single* document **must** include the *shard key* or the `_id` field. Updates that affect multiple documents are more efficient in some situations if they have the *shard key*, but can be broadcast to all shards.

If the value of the shard key increases or decreases with every insert, all insert operations target a single shard. As a result, the capacity of a single shard becomes the limit for the insert capacity of the sharded cluster.

For more information, see [Sharded Cluster Tutorials](#) (page 513) and [Bulk Inserts in MongoDB](#) (page 53).

Write Operations on Replica Sets In *replica sets*, all write operations go to the set's *primary*, which applies the write operation then records the operations on the primary's operation log or *oplog*. The oplog is a reproducible sequence of operations to the data set. *Secondary* members of the set are continuously replicating the oplog and applying the operations to themselves in an asynchronous process.

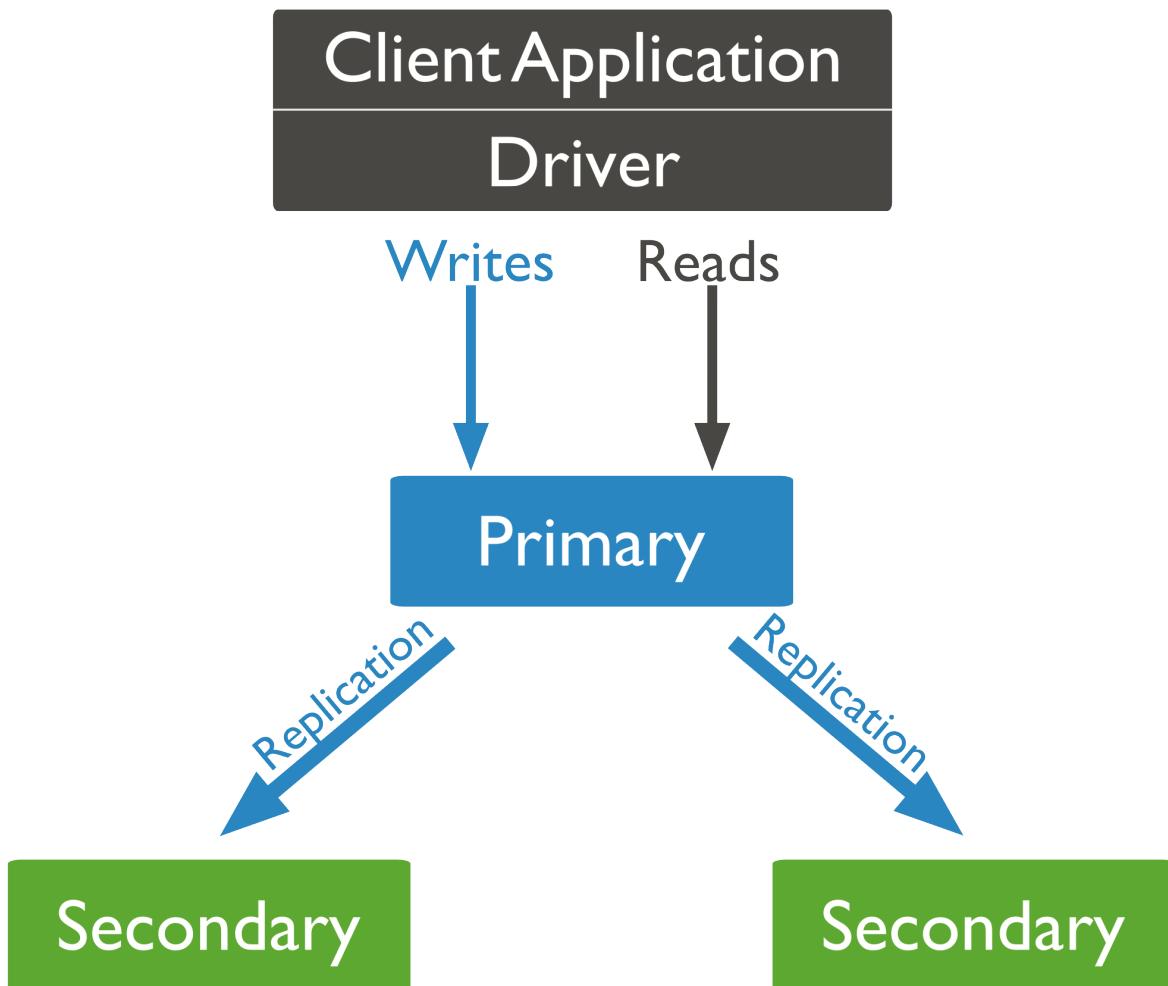


Figure 2.25: Diagram of default routing of reads and writes to the primary.

Large volumes of write operations, particularly bulk operations, may create situations where the secondary members have difficulty applying the replicating operations from the primary at a sufficient rate: this can cause the secondary's state to fall behind that of the primary. Secondaries that are significantly behind the primary present problems for

normal operation of the replica set, particularly [failover](#) (page 392) in the form of [rollbacks](#) (page 397) as well as general [read consistency](#) (page 398).

To help avoid this issue, you can customize the [write concern](#) (page 44) to return confirmation of the write operation to another member⁴ of the replica set every 100 or 1,000 operations. This provides an opportunity for secondaries to catch up with the primary. Write concern can slow the overall progress of write operations but ensure that the secondaries can maintain a largely current state with respect to the primary.

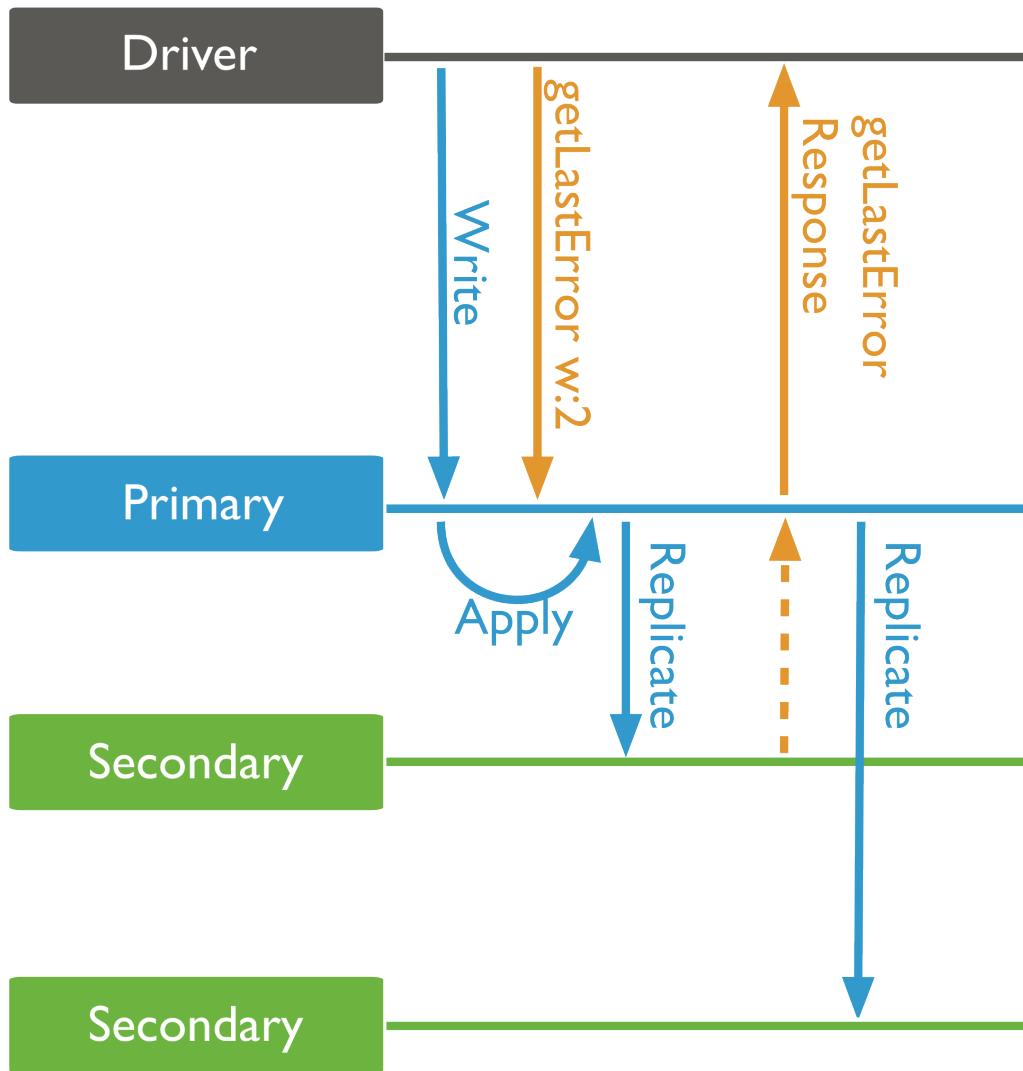


Figure 2.26: Write operation to a replica set with write concern level of `w:2` or write to the primary and at least one secondary.

For more information on replica sets and write operations, see [Replica Acknowledged](#) (page 47), [Oplog Size](#) (page 406), and [Change the Size of the Oplog](#) (page 441).

⁴ Calling `getLastError` intermittently with a `w` value of 2 or `majority` will slow the throughput of write traffic; however, this practice will allow the secondaries to remain current with the state of the primary.

Write Operation Performance

- [Indexes \(page 52\)](#)
- [Document Growth \(page 52\)](#)
- [Storage Performance \(page 52\)](#)
 - [Hardware \(page 52\)](#)
 - [Journaling \(page 52\)](#)

Indexes After every insert, update, or delete operation, MongoDB must update *every* index associated with the collection in addition to the data itself. Therefore, every index on a collection adds some amount of overhead for the performance of write operations.⁵

In general, the performance gains that indexes provide for *read operations* are worth the insertion penalty. However, in order to optimize write performance when possible, be careful when creating new indexes and evaluate the existing indexes to ensure that your queries actually use these indexes.

For indexes and queries, see [Query Optimization \(page 34\)](#). For more information on indexes, see [Indexes \(page 309\)](#) and [Indexing Strategies \(page 363\)](#).

Document Growth If an update operation causes a document to exceed the currently allocated *record size*, MongoDB relocates the document on disk with enough contiguous space to hold the document. These relocations take longer than in-place updates, particularly if the collection has indexes. If a collection has indexes, MongoDB must update all index entries. Thus, for a collection with many indexes, the move will impact the write throughput.

Some update operations, such as the `$inc` operation, do not cause an increase in document size. For these update operations, MongoDB can apply the updates in-place. Other update operations, such as the `$push` operation, change the size of the document.

In-place-updates are significantly more efficient than updates that cause document growth. When possible, use [data models \(page 97\)](#) that minimize the need for document growth.

See [Record Padding \(page 54\)](#) for more information.

Storage Performance

Hardware The capability of the storage system creates some important physical limits for the performance of MongoDB's write operations. Many unique factors related to the storage system of the drive affect write performance, including random access patterns, disk caches, disk readahead and RAID configurations.

Solid state drives (SSDs) can outperform spinning hard disks (HDDs) by 100 times or more for random workloads.

See

[Production Notes \(page 151\)](#) for recommendations regarding additional hardware and configuration options.

Journaling MongoDB uses *write ahead logging* to an on-disk *journal* to guarantee [write operation \(page 40\)](#) durability and to provide crash resiliency. Before applying a change to the data files, MongoDB writes the change operation to the journal.

⁵ For inserts and updates to un-indexed fields, the overhead for [sparse indexes \(page 331\)](#) is less than for non-sparse indexes. Also for non-sparse indexes, updates that do not change the record size have less indexing overhead.

While the durability assurance provided by the journal typically outweigh the performance costs of the additional write operations, consider the following interactions between the journal and performance:

- if the journal and the data file reside on the same block device, the data files and the journal may have to contend for a finite number of available write operations. Moving the journal to a separate device may increase the capacity for write operations.
- if applications specify [write concern](#) (page 44) that includes [journalized](#) (page 45), mongod will decrease the duration between journal commits, which can increase the overall write load.
- the duration between journal commits is configurable using the `journalCommitInterval` run-time option. Decreasing the period between journal commits will increase the number of write operations, which can limit MongoDB's capacity for write operations. Increasing the amount of time between commits may decrease the total number of write operations, but also increases the chance that the journal will not record a write operation in the event of a failure.

For additional information on journaling, see [Journaling Mechanics](#) (page 229).

Bulk Inserts in MongoDB

- [Use the `insert\(\)` Method](#) (page 53)
- [Bulk Inserts on Sharded Clusters](#) (page 53)
 - [Pre-Split the Collection](#) (page 54)
 - [Insert to Multiple `mongos`](#) (page 54)
 - [Avoid Monotonic Throttling](#) (page 54)

In some situations you may need to insert or ingest a large amount of data into a MongoDB database. These *bulk inserts* have some special considerations that are different from other write operations.

Use the `insert()` Method The `insert()` method, when passed an array of documents, performs a bulk insert, and inserts each document atomically. Bulk inserts can significantly increase performance by amortizing [write concern](#) (page 44) costs.

New in version 2.2: `insert()` in the mongo shell gained support for bulk inserts in version 2.2.

In the [drivers](#) (page 92), you can configure write concern for batches rather than on a per-document level.

Drivers have a `ContinueOnError` option in their `insert` operation, so that the bulk operation will continue to insert remaining documents in a batch even if an insert fails.

Note: If multiple errors occur during a bulk insert, clients only receive the last error generated.

See also:

[Driver documentation](#) (page 92) for details on performing bulk inserts in your application. Also see [Import and Export MongoDB Data](#) (page 147).

Bulk Inserts on Sharded Clusters While `ContinueOnError` is optional on unsharded clusters, all bulk operations to a *sharded collection* run with `ContinueOnError`, which cannot be disabled.

Large bulk insert operations, including initial data inserts or routine data import, can affect *sharded cluster* performance. For bulk inserts, consider the following strategies:

Pre-Split the Collection If the sharded collection is empty, then the collection has only one initial *chunk*, which resides on a single shard. MongoDB must then take time to receive data, create splits, and distribute the split chunks to the available shards. To avoid this performance cost, you can pre-split the collection, as described in [Split Chunks in a Sharded Cluster](#) (page 545).

Insert to Multiple mongos To parallelize import processes, send insert operations to more than one mongos instance. Pre-split empty collections first as described in [Split Chunks in a Sharded Cluster](#) (page 545).

Avoid Monotonic Throttling If your shard key increases monotonically during an insert, then all inserted data goes to the last chunk in the collection, which will always end up on a single shard. Therefore, the insert capacity of the cluster will never exceed the insert capacity of that single shard.

If your insert volume is larger than what a single shard can process, and if you cannot avoid a monotonically increasing shard key, then consider the following modifications to your application:

- Reverse the binary bits of the shard key. This preserves the information and avoids correlating insertion order with increasing sequence of values.
- Swap the first and last 16-bit words to “shuffle” the inserts.

Example

The following example, in C++, swaps the leading and trailing 16-bit word of *BSON ObjectIds* generated so that they are no longer monotonically increasing.

```
using namespace mongo;
OID make_an_id() {
    OID x = OID::gen();
    const unsigned char *p = x.getData();
    swap( (unsigned short&) p[0], (unsigned short&) p[10] );
    return x;
}

void foo() {
    // create an object
    BSONObj o = BSON( "_id" << make_an_id() << "x" << 3 << "name" << "jane" );
    // now we may insert o into a sharded collection
}
```

See also:

[Shard Keys](#) (page 499) for information on choosing a sharded key. Also see [Shard Key Internals](#) (page 499) (in particular, [Choosing a Shard Key](#) (page 518)).

Record Padding

Update operations can increase the size of the document⁶. If a document outgrows its current allocated *record space*, MongoDB must allocate a new space and move the document to this new location.

To reduce the number of moves, MongoDB includes a small amount of extra space, or *padding*, when allocating the record space. This padding reduces the likelihood that a slight increase in document size will cause the document to exceed its allocated record size.

See also:

[Write Operation Performance](#) (page 52).

⁶ Documents in MongoDB can grow up to the full maximum BSON document size.

Padding Factor To minimize document movements and their impact, MongoDB employs padding. MongoDB adaptively adjusts the size of record allocations in a collection by adding a `paddingFactor` so that the documents have room to grow. The `paddingFactor` indicates the padding for new inserts and moves.

To check the current `paddingFactor` on a collection, you can run the `db.collection.stats()` operation in the mongo shell, as in the following example:

```
db.myCollection.stats()
```

Since MongoDB writes each document at a different point in time, the padding for each document will not be the same. You can calculate the padding size by subtracting 1 from the `paddingFactor`, for example:

```
padding size = (paddingFactor - 1) * <document size>.
```

For example, a `paddingFactor` of `1.0` specifies no padding whereas a `paddingFactor` of `1.5` specifies a padding size of `0.5` or 50 percent (50%) of the document size.

Because the `paddingFactor` is relative to the size of each document, you cannot calculate the exact amount of padding for a collection based on the average document size and padding factor.

If an update operation causes the document to *decrease* in size, for instance if you perform an `$unset` or a `$pop` update, the document remains in place and effectively has more padding. If the document remains this size, the space is not reclaimed until you perform a `compact` or a `repairDatabase` operation.

Operations That Remove Padding The following operations remove padding: `compact`, `repairDatabase`, and initial replica sync operations. However, with the `compact` command, you can run the command with a `paddingFactor` or a `paddingBytes` parameter. See `compact` command for details.

Padding is also removed if you use `mongoexport` a collection. If you use `mongoimport` into a new collection, `mongoimport` will not add padding. If you use `mongoimport` with an existing collection with padding, `mongoimport` will not affect the existing padding.

When a database operation removes padding from a collection, subsequent updates to the collection that increase the record size will have reduced throughput until the collection's padding factor grows. However, the collection will require less storage.

Record Allocation Strategies New in version 2.2: `collMod` and `usePowerOf2Sizes`.

To more efficiently reuse the space freed as a result of deletions or document relocations, you can specify that MongoDB allocates record sizes in powers of 2. To do so, use the `collMod` command with the `usePowerOf2Sizes` flag. See `collMod` command for more details. As with all padding, power of 2 size allocations minimizes, but does not eliminate, document movements.

See also [Can I manually pad documents to prevent moves during updates?](#) (page 583)

2.3 MongoDB CRUD Tutorials

The following tutorials provide instructions for querying and modifying data. For a higher-level overview of these operations, see [MongoDB CRUD Operations](#) (page 25).

[**Insert Documents**](#) (page 56) Insert new documents into a collection.

[**Query Documents**](#) (page 57) Find documents in a collection using search criteria.

[**Limit Fields to Return from a Query**](#) (page 61) Limit which fields are returned by a query.

[**Iterate a Cursor in the mongo Shell**](#) (page 62) Access documents returned by a `find` query by iterating the cursor, either manually or using the iterator index.

[Analyze Query Performance \(page 63\)](#) Analyze the efficiency of queries and determine how a query uses available indexes.

[Modify Documents \(page 64\)](#) Modify documents in a collection

[Remove Documents \(page 65\)](#) Remove documents from a collection.

[Perform Two Phase Commits \(page 66\)](#) Use two-phase commits when writing data to multiple documents.

[Create Tailable Cursor \(page 71\)](#) Create tailable cursors for use in capped collections with high numbers of write operations for which an index would be too expensive.

[Isolate Sequence of Operations \(page 74\)](#) Use the `<isolation>` `isolated` operator to *isolate* a single write operation that affects multiple documents, preventing other operations from interrupting the sequence of write operations.

[Create an Auto-Incrementing Sequence Field \(page 75\)](#) Describes how to create an incrementing sequence number for the `_id` field using a Counters Collection or an Optimistic Loop.

[Limit Number of Elements in an Array after an Update \(page 78\)](#) Use `$push` with various modifiers to sort and maintain an array of fixed size after update

2.3.1 Insert Documents

In MongoDB, the `db.collection.insert()` method adds new documents into a collection. In addition, both the `db.collection.update()` method and the `db.collection.save()` method can also add new documents through an operation called an *upsert*. An *upsert* is an operation that performs either an update of an existing document or an insert of a new document if the document to modify does not exist.

This tutorial provides examples of insert operations using each of the three methods in the mongo shell.

Insert a Document with `insert()` Method

The following statement inserts a document with three fields into the collection `inventory`:

```
db.inventory.insert( { _id: 10, type: "misco", item: "card", qty: 15 } )
```

In the example, the document has a user-specified `_id` field value of 10. The value must be unique within the `inventory` collection.

For more examples, see `insert()`.

Insert a Document with `update()` Method

Call the `update()` method with the `upsert` flag to create a new document if no document matches the update's query criteria.⁷

The following example creates a new document if no document in the `inventory` collection contains `{ type: "books", item : "journal" }`:

```
db.inventory.update(
    { type: "book", item : "journal" },
    { $set : { qty: 10 } },
    { upsert : true }
)
```

⁷ Prior to version 2.2, in the mongo shell, you would specify the `upsert` and the `multi` options in the `update()` method as positional boolean options. See `update()` for details.

MongoDB adds the `_id` field and assigns as its value a unique ObjectId. The new document includes the `item` and `type` fields from the `<query>` criteria and the `qty` field from the `<update>` parameter.

```
{ "_id" : ObjectId("51e8636953dbe31d5f34a38a"), "item" : "journal", "qty" : 10, "type" : "book" }
```

For more examples, see `update()`.

Insert a Document with `save()` Method

To insert a document with the `save()` method, pass the method a document that does not contain the `_id` field or a document that contains an `_id` field that does not exist in the collection.

The following example creates a new document in the `inventory` collection:

```
db.inventory.save( { type: "book", item: "notebook", qty: 40 } )
```

MongoDB adds the `_id` field and assigns as its value a unique ObjectId.

```
{ "_id" : ObjectId("51e866e48737f72b32ae4fbc"), "type" : "book", "item" : "notebook", "qty" : 40 }
```

For more examples, see `save()`.

2.3.2 Query Documents

In MongoDB, the `db.collection.find()` method retrieves documents from a collection.⁸ The `db.collection.find()` method returns a *cursor* (page 33) to the retrieved documents.

This tutorial provides examples of read operations using the `db.collection.find()` method in the mongo shell. In these examples, the retrieved documents contain all their fields. To restrict the fields to return in the retrieved documents, see [Limit Fields to Return from a Query](#) (page 61).

Select All Documents in a Collection

An empty query document (`{ }`) selects all documents in the collection:

```
db.inventory.find( {} )
```

Not specifying a query document to the `find()` is equivalent to specifying an empty query document. Therefore the following operation is equivalent to the previous operation:

```
db.inventory.find()
```

Specify Equality Condition

To specify equality condition, use the query document `{ <field>: <value> }` to select all documents that contain the `<field>` with the specified `<value>`.

The following example retrieves from the `inventory` collection all documents where the `type` field has the value `snacks`:

```
db.inventory.find( { type: "snacks" } )
```

⁸ The `db.collection.findOne()` method also performs a read operation to return a single document. Internally, the `db.collection.findOne()` method is the `db.collection.find()` method with a limit of 1.

Specify Conditions Using Query Operators

A query document can use the *query operators* to specify conditions in a MongoDB query.

The following example selects all documents in the `inventory` collection where the value of the `type` field is either `'food'` or `'snacks'`:

```
db.inventory.find( { type: { $in: [ 'food', 'snacks' ] } } )
```

Although you can express this query using the `$or` operator, use the `$in` operator rather than the `$or` operator when performing equality checks on the same field.

Refer to the <http://docs.mongodb.org/manual/reference/operator> document for the complete list of query operators.

Specify AND Conditions

A compound query can specify conditions for more than one field in the collection's documents. Implicitly, a logical `AND` conjunction connects the clauses of a compound query so that the query selects the documents in the collection that match all the conditions.

In the following example, the query document specifies an equality match on the field `food` **and** a less than (`$lt`) comparison match on the field `price`:

```
db.inventory.find( { type: 'food', price: { $lt: 9.95 } } )
```

This query selects all documents where the `type` field has the value `'food'` **and** the value of the `price` field is less than `9.95`. See *comparison operators* for other comparison operators.

Specify OR Conditions

Using the `$or` operator, you can specify a compound query that joins each clause with a logical `OR` conjunction so that the query selects the documents in the collection that match at least one condition.

In the following example, the query document selects all documents in the collection where the field `qty` has a value greater than (`$gt`) `100` **or** the value of the `price` field is less than (`$lt`) `9.95`:

```
db.inventory.find(
  { $or: [
    { qty: { $gt: 100 } },
    { price: { $lt: 9.95 } }
  ]
}
```

Specify AND as well as OR Conditions

With additional clauses, you can specify precise conditions for matching documents.

In the following example, the compound query document selects all documents in the collection where the value of the `type` field is `'food'` **and** *either* the `qty` has a value greater than (`$gt`) `100` *or* the value of the `price` field is less than (`$lt`) `9.95`:

```
db.inventory.find( { type: 'food', $or: [ { qty: { $gt: 100 } },
                                         { price: { $lt: 9.95 } } ]
} )
```

Subdocuments

When the field holds an embedded document (i.e. subdocument), you can either specify the entire subdocument as the value of a field, or “reach into” the subdocument using *dot notation*, to specify values for individual fields in the subdocument:

Exact Match on Subdocument

To specify an equality match on the whole subdocument, use the query document { <field>: <value> } where <value> is the subdocument to match. Equality matches on a subdocument require that the subdocument field match *exactly* the specified <value>, including the field order.

In the following example, the query matches all documents where the value of the field `producer` is a subdocument that contains *only* the field `company` with the value '`ABC123`' and the field `address` with the value '`123 Street`', in the exact order:

```
db.inventory.find(
  {
    producer: {
      company: 'ABC123',
      address: '123 Street'
    }
  }
)
```

Equality Match on Fields within Subdocument

Equality matches for specific fields within subdocuments select the documents in the collection when the field in the subdocument contains a field that matches the specified value.

In the following example, the query uses the *dot notation* to match all documents where the value of the field `producer` is a subdocument that contains a field `company` with the value '`ABC123`' and may contain other fields:

```
db.inventory.find( { 'producer.company': 'ABC123' } )
```

Arrays

When the field holds an array, you can query for an exact array match or for specific values in the array. If the array holds sub-documents, you can query for specific fields within the sub-documents using *dot notation*:

Exact Match on an Array

To specify equality match on an array, use the query document { <field>: <value> } where <value> is the array to match. Equality matches on the array require that the array field match *exactly* the specified <value>, including the element order.

In the following example, the query matches all documents where the value of the field `tags` is an array that holds exactly three elements, '`fruit`', '`food`', and '`citrus`', in this order:

```
db.inventory.find( { tags: [ 'fruit', 'food', 'citrus' ] } )
```

Match an Array Element

Equality matches can specify a single element in the array to match. These specifications match if the array contains at least *one* element with the specified value.

In the following example, the query matches all documents where the value of the field `tags` is an array that contains '`fruit`' as one of its elements:

```
db.inventory.find( { tags: 'fruit' } )
```

Match a Specific Element of an Array

Equality matches can specify equality matches for an element at a particular index or position of the array.

In the following example, the query uses the *dot notation* to match all documents where the value of the `tags` field is an array whose first element equals '`fruit`':

```
db.inventory.find( { 'tags.0' : 'fruit' } )
```

Array of Subdocuments

Match a Field in the Subdocument Using the Array Index If you know the array index of the subdocument, you can specify the document using the subdocument's position.

The following example selects all documents where the `memos` contains an array whose first element (i.e. index is 0) is a subdocument with the field `by` with the value '`shipping`':

```
db.inventory.find( { 'memos.0.by': 'shipping' } )
```

Match a Field Without Specifying Array Index If you do not know the index position of the subdocument, concatenate the name of the field that contains the array, with a dot (.) and the name of the field in the subdocument.

The following example selects all documents where the `memos` field contains an array that contains at least one subdocument with the field `by` with the value '`shipping`':

```
db.inventory.find( { 'memos.by': 'shipping' } )
```

Match Multiple Fields To match by multiple fields in the subdocument, you can use either dot notation or the `$elemMatch` operator:

The following example uses dot notation to query for documents where the value of the `memos` field is an array that has at least one subdocument that contains the field `memo` equal to '`on time`' and the field `by` equal to '`shipping`':

```
db.inventory.find(
  {
    'memos.memo': 'on time',
    'memos.by': 'shipping'
  }
)
```

The following example uses `$elemMatch` to query for documents where the value of the `memos` field is an array that has at least one subdocument that contains the field `memo` equal to '`on time`' and the field `by` equal to '`shipping`':

```
db.inventory.find( {
    memos: {
        $elemMatch: {
            memo : 'on time',
            by: 'shipping'
        }
    }
})
```

2.3.3 Limit Fields to Return from a Query

The *projection* specification limits the fields to return for all matching documents. The projection takes the form of a *document* with a list of fields for inclusion or exclusion from the result set. You can either specify the fields to include (e.g. { *field*: 1 }) or specify the fields to exclude (e.g. { *field*: 0 }).

Important: The `_id` field is, by default, included in the result set. To exclude the `_id` field from the result set, you need to specify in the projection document the exclusion of the `_id` field (i.e. { `_id`: 0 }).

You cannot combine inclusion and exclusion semantics in a single projection with the *exception* of the `_id` field.

This tutorial offers various query examples that limit the fields to return for all matching documents. The examples in this tutorial use a collection `inventory` and use the `db.collection.find()` method in the mongo shell. The `db.collection.find()` method returns a [cursor](#) (page 33) to the retrieved documents. For examples on query selection criteria, see [Query Documents](#) (page 57).

Return All Fields in Matching Documents

If you specify no projection, the `find()` method returns all fields of all documents that match the query.

```
db.inventory.find( { type: 'food' } )
```

This operation will return all documents in the `inventory` collection where the value of the `type` field is ' food'. The returned documents contain all its fields.

Return the Specified Fields and the `_id` Field Only

A projection can explicitly include several fields. In the following operation, `find()` method returns all documents that match the query. In the result set, only the `item` and `qty` fields and, by default, the `_id` field return in the matching documents.

```
db.inventory.find( { type: 'food' }, { item: 1, qty: 1 } )
```

Return Specified Fields Only

You can remove the `_id` field from the results by specifying its exclusion in the projection, as in the following example:

```
db.inventory.find( { type: 'food' }, { item: 1, qty: 1, _id:0 } )
```

This operation returns all documents that match the query. In the result set, *only* the `item` and `qty` fields return in the matching documents.

Return All But the Excluded Field

To exclude a single field or group of fields you can use a projection in the following form:

```
db.inventory.find( { type: 'food' }, { type:0 } )
```

This operation returns all documents where the value of the `type` field is `food`. In the result set, the `type` field does not return in the matching documents.

With the exception of the `_id` field you cannot combine inclusion and exclusion statements in projection documents.

Projection for Array Fields

The `$elemMatch` and `$slice` projection operators provide more control when projecting only a portion of an array.

2.3.4 Iterate a Cursor in the mongo Shell

The `db.collection.find()` method returns a cursor. To access the documents, you need to iterate the cursor. 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 in the results. The following describes ways to manually iterate the cursor to access the documents or to use the iterator index.

Manually Iterate the Cursor

In the mongo shell, when you assign the cursor returned from the `find()` method to a variable using the `var` keyword, the cursor does not automatically iterate.

You can call the cursor variable in the shell to iterate up to 20 times ⁹ and print the matching documents, as in the following example:

```
var myCursor = db.inventory.find( { type: 'food' } );  
  
myCursor
```

You can also use the cursor method `next()` to access the documents, as in the following example:

```
var myCursor = db.inventory.find( { type: 'food' } );  
var myDocument = myCursor.hasNext() ? myCursor.next() : null;  
  
if (myDocument) {  
    var myItem = myDocument.item;  
    print(tojson(myItem));  
}
```

As an alternative print operation, consider the `printjson()` helper method to replace `print(tojson())`:

```
if (myDocument) {  
    var myItem = myDocument.item;  
    printjson(myItem);  
}
```

You can use the cursor method `forEach()` to iterate the cursor and access the documents, as in the following example:

⁹ You can use the `DBQuery.shellBatchSize` to change the number of iteration from the default value 20. See [Executing Queries](#) (page 209) for more information.

```
var myCursor = db.inventory.find( { type: 'food' } );
myCursor.forEach(printjson);
```

See *JavaScript cursor methods* and your *driver* (page 92) documentation for more information on cursor methods.

Iterator Index

In the mongo shell, you can use the `toArray()` method to iterate the cursor and return the documents in an array, as in the following:

```
var myCursor = db.inventory.find( { type: 'food' } );
var documentArray = myCursor.toArray();
var myDocument = documentArray[3];
```

The `toArray()` method loads into RAM all documents returned by the cursor; the `toArray()` method exhausts the cursor.

Additionally, some *drivers* (page 92) provide access to the documents by using an index on the cursor (i.e. `cursor[index]`). This is a shortcut for first calling the `toArray()` method and then using an index on the resulting array.

Consider the following example:

```
var myCursor = db.inventory.find( { type: 'food' } );
var myDocument = myCursor[3];
```

The `myCursor[3]` is equivalent to the following example:

```
myCursor.toArray() [3];
```

2.3.5 Analyze Query Performance

The `explain()` cursor method allows you to inspect the operation of the query system. This method is useful for analyzing the efficiency of queries, and for determining how the query uses the index. The `explain()` method tests the query operation, and *not* the timing of query performance. Because `explain()` attempts multiple query plans, it does not reflect an accurate timing of query performance.

Evaluate the Performance of a Query

To use the `explain()` method, call the method on a cursor returned by `find()`.

Example

Evaluate a query on the `type` field on the collection `inventory` that has an index on the `type` field.

```
db.inventory.find( { type: 'food' } ).explain()
```

Consider the results:

```
{
  "cursor" : "BtreeCursor type_1",
  "isMultiKey" : false,
  "n" : 5,
  "nscannedObjects" : 5,
  "nscanned" : 5,
```

```
"nscannedObjectsAllPlans" : 5,
"nscannedAllPlans" : 5,
"scanAndOrder" : false,
"indexOnly" : false,
"nYields" : 0,
"nChunkSkips" : 0,
"millis" : 0,
"indexBounds" : { "type" : [
    [ "food",
      "food" ]
  ],
"server" : "mongodb0.example.net:27017" }
```

The `BtreeCursor` value of the `cursor` field indicates that the query used an index.

This query returned 5 documents, as indicated by the `n` field.

To return these 5 documents, the query scanned 5 documents from the index, as indicated by the `nscanned` field, and then read 5 full documents from the collection, as indicated by the `nscannedObjects` field.

Without the index, the query would have scanned the whole collection to return the 5 documents.

See *explain-results* method for full details on the output.

Compare Performance of Indexes

To manually compare the performance of a query using more than one index, you can use the `hint()` and `explain()` methods in conjunction.

Example

Evaluate a query using different indexes:

```
db.inventory.find( { type: 'food' } ).hint( { type: 1 } ).explain()
db.inventory.find( { type: 'food' } ).hint( { type: 1, name: 1 } ).explain()
```

These return the statistics regarding the execution of the query using the respective index.

Note: If you run `explain()` without including `hint()`, the query optimizer reevaluates the query and runs against multiple indexes before returning the query statistics.

For more detail on the `explain` output, see *explain-results*.

2.3.6 Modify Documents

In MongoDB, both `db.collection.update()` and `db.collection.save()` modify existing documents in a collection. `db.collection.update()` provides additional control over the modification. For example, you can modify existing data or modify a group of documents that match a query with `db.collection.update()`. Alternately, `db.collection.save()` replaces an existing document with the same `_id` field.

This document provides examples of the update operations using each of the two methods in the `mongo` shell.

Modify Multiple Documents with `update()` Method

By default, the `update()` method updates a single document that matches its selection criteria. Call the method with the `multi` option set to `true` to update multiple documents.¹⁰

The following example finds all documents with `type` equal to "book" and modifies their `qty` field by `-1`. The example uses `$inc`, which is one of the:`update operators <update-operators>` available.

```
db.inventory.update(
  { type : "book" },
  { $inc : { qty : -1 } },
  { multi: true }
)
```

For more examples, see `update()`.

Modify a Document with `save()` Method

The `save()` method can replace an existing document. To replace a document with the `save()` method, pass the method a document with an `_id` field that matches an existing document.

The following example completely replaces the document with the `_id` equal to `10` in the `inventory` collection:

```
db.inventory.save(
{
  _id: 10,
  type: "misc",
  item: "placard"
}
)
```

For further examples, see `save()`.

2.3.7 Remove Documents

In MongoDB, the `db.collection.remove()` method removes documents from a collection. You can remove all documents, specify which documents to remove, and limit the operation to a single document.

This tutorial provides examples of remove operations using the `db.collection.remove()` method in the mongo shell.

Remove All Documents

If you do not specify a query, `remove()` removes all documents from a collection, but does not remove the indexes.¹¹

The following example removes all documents from the `inventory` collection:

```
db.inventory.remove()
```

¹⁰ This shows the syntax for MongoDB 2.2 and later. For syntax for versions prior to 2.2, see `update()`.

¹¹ To remove all documents from a collection, it may be more efficient to use the `drop()` method to drop the entire collection, including the indexes, and then recreate the collection and rebuild the indexes.

Remove Documents that Matches a Condition

To remove the documents that match a deletion criteria, call the `remove()` method with the `<query>` parameter.

The following example removes all documents that have `type` equal to `food` from the `inventory` collection:

```
db.inventory.remove( { type : "food" } )
```

Note: For large deletion operations, it may be more efficient to copy the documents that you want to keep to a new collection and then use `drop()` on the original collection.

Remove a Single Document that Matches a Condition

To remove a single document, call the `remove()` method with the `justOne` parameter set to `true` or `1`.

The following example removes one document that have `type` equal to `food` from the `inventory` collection:

```
db.inventory.remove( { type : "food" }, 1 )
```

2.3.8 Perform Two Phase Commits

Synopsis

This document provides a pattern for doing multi-document updates or “transactions” using a two-phase commit approach for writing data to multiple documents. Additionally, you can extend this process to provide a *rollback* (page 69) like functionality.

Background

Operations on a single *document* are always atomic with MongoDB databases; however, operations that involve multiple documents, which are often referred to as “transactions,” are not atomic. Since documents can be fairly complex and contain multiple “nested” documents, single-document atomicity provides necessary support for many practical use cases.

Thus, without precautions, success or failure of the database operation cannot be “all or nothing,” and without support for multi-document transactions it’s possible for an operation to succeed for some operations and fail with others. When executing a transaction composed of several sequential operations the following issues arise:

- Atomicity: if one operation fails, the previous operation within the transaction must “rollback” to the previous state (i.e. the “nothing,” in “all or nothing.”)
- Isolation: operations that run concurrently with the transaction operation set must “see” a consistent view of the data throughout the transaction process.
- Consistency: if a major failure (i.e. network, hardware) interrupts the transaction, the database must be able to recover a consistent state.

Despite the power of single-document atomic operations, there are cases that require multi-document transactions. For these situations, you can use a two-phase commit, to provide support for these kinds of multi-document updates.

Because documents can represent both pending data and states, you can use a two-phase commit to ensure that data is consistent, and that in the case of an error, the state that preceded the transaction is *recoverable* (page 69).

Note: Because only single-document operations are atomic with MongoDB, two-phase commits can only offer

transaction-*like* semantics. It's possible for applications to return intermediate data at intermediate points during the two-phase commit or rollback.

Pattern

Overview

The most common example of transaction is to transfer funds from account A to B in a reliable way, and this pattern uses this operation as an example. In a relational database system, this operation would encapsulate subtracting funds from the source (A) account and adding them to the destination (B) within a single atomic transaction. For MongoDB, you can use a two-phase commit in these situations to achieve a compatible response.

All of the examples in this document use the mongo shell to interact with the database, and assume that you have two collections: First, a collection named `accounts` that will store data about accounts with one account per document, and a collection named `transactions` which will store the transactions themselves.

Begin by creating two accounts named A and B, with the following command:

```
db.accounts.save({name: "A", balance: 1000, pendingTransactions: []})
db.accounts.save({name: "B", balance: 1000, pendingTransactions: []})
```

To verify that these operations succeeded, use `find()`:

```
db.accounts.find()
```

`mongo` will return two *documents* that resemble the following:

```
{ "_id" : ObjectId("4d7bc66cb8a04f512696151f"), "name" : "A", "balance" : 1000, "pendingTransactions": []}
{ "_id" : ObjectId("4d7bc67bb8a04f5126961520"), "name" : "B", "balance" : 1000, "pendingTransactions": []}
```

Transaction Description

Set Transaction State to Initial Create the `transaction` collection by inserting the following document. The transaction document holds the `source` and `destination`, which refer to the `name` fields of the `accounts` collection, as well as the `value` field that represents the amount of data change to the `balance` field. Finally, the `state` field reflects the current state of the transaction.

```
db.transactions.save({source: "A", destination: "B", value: 100, state: "initial"})
```

To verify that these operations succeeded, use `find()`:

```
db.transactions.find()
```

This will return a document similar to the following:

```
{ "_id" : ObjectId("4d7bc7a8b8a04f5126961522"), "source" : "A", "destination" : "B", "value" : 100, "state" : "initial"}
```

Switch Transaction State to Pending Before modifying either records in the `accounts` collection, set the transaction state to `pending` from `initial`.

Set the local variable `t` in your shell session, to the transaction document using `findOne()`:

```
t = db.transactions.findOne({state: "initial"})
```

After assigning this variable `t`, the shell will return the value of `t`, you will see the following output:

```
{  
  "_id" : ObjectId("4d7bc7a8b8a04f5126961522"),  
  "source" : "A",  
  "destination" : "B",  
  "value" : 100,  
  "state" : "initial"  
}
```

Use `update()` to change the value of `state` to `pending`:

```
db.transactions.update({_id: t._id}, {$set: {state: "pending"}})  
db.transactions.find()
```

The `find()` operation will return the contents of the `transactions` collection, which should resemble the following:

```
{ "_id" : ObjectId("4d7bc7a8b8a04f5126961522"), "source" : "A", "destination" : "B", "value" : 100, "state" : "pending" }
```

Apply Transaction to Both Accounts Continue by applying the transaction to both accounts. The `update()` query will prevent you from applying the transaction *if* the transaction is *not* already pending. Use the following `update()` operation:

```
db.accounts.update({name: t.source, pendingTransactions: {$ne: t._id}}, {$inc: {balance: -t.value}}, {upsert: true})  
db.accounts.update({name: t.destination, pendingTransactions: {$ne: t._id}}, {$inc: {balance: t.value}}, {upsert: true})  
db.accounts.find()
```

The `find()` operation will return the contents of the `accounts` collection, which should now resemble the following:

```
{ "_id" : ObjectId("4d7bc97fb8a04f5126961523"), "balance" : 900, "name" : "A", "pendingTransactions" : 100 }  
{ "_id" : ObjectId("4d7bc984b8a04f5126961524"), "balance" : 1100, "name" : "B", "pendingTransactions" : 100 }
```

Set Transaction State to Committed Use the following `update()` operation to set the transaction's state to `committed`:

```
db.transactions.update({_id: t._id}, {$set: {state: "committed"}})  
db.transactions.find()
```

The `find()` operation will return the contents of the `transactions` collection, which should now resemble the following:

```
{ "_id" : ObjectId("4d7bc7a8b8a04f5126961522"), "destination" : "B", "source" : "A", "state" : "committed" }
```

Remove Pending Transaction Use the following `update()` operation to set remove the pending transaction from the `documents` in the `accounts` collection:

```
db.accounts.update({name: t.source}, {$pull: {pendingTransactions: t._id}})  
db.accounts.update({name: t.destination}, {$pull: {pendingTransactions: t._id}})  
db.accounts.find()
```

The `find()` operation will return the contents of the `accounts` collection, which should now resemble the following:

```
{ "_id" : ObjectId("4d7bc97fb8a04f5126961523"), "balance" : 900, "name" : "A", "pendingTransactions" : 0 }  
{ "_id" : ObjectId("4d7bc984b8a04f5126961524"), "balance" : 1100, "name" : "B", "pendingTransactions" : 0 }
```

Set Transaction State to Done Complete the transaction by setting the `state` of the transaction *document* to `done`:

```
db.transactions.update({_id: t._id}, {$set: {state: "done"})})
db.transactions.find()
```

The `find()` operation will return the contents of the `transactions` collection, which should now resemble the following:

```
{ "_id" : ObjectId("4d7bc7a8b8a04f5126961522"), "destination" : "B", "source" : "A", "state" : "done"}
```

Recovering from Failure Scenarios

The most important part of the transaction procedure is not, the prototypical example above, but rather the possibility for recovering from the various failure scenarios when transactions do not complete as intended. This section will provide an overview of possible failures and provide methods to recover from these kinds of events.

There are two classes of failures:

- all failures that occur after the first step (i.e. [setting the transaction set to initial](#) (page 67)) but before the third step (i.e. [applying the transaction to both accounts](#) (page 68).)

To recover, applications should get a list of transactions in the `pending` state and resume from the second step (i.e. [switching the transaction state to pending](#) (page 67).)

- all failures that occur after the third step (i.e. [applying the transaction to both accounts](#) (page 68)) but before the fifth step (i.e. [setting the transaction state to done](#) (page 68).)

To recover, application should get a list of transactions in the `committed` state and resume from the fourth step (i.e. [remove the pending transaction](#) (page 68).)

Thus, the application will always be able to resume the transaction and eventually arrive at a consistent state. Run the following recovery operations every time the application starts to catch any unfinished transactions. You may also wish run the recovery operation at regular intervals to ensure that your data remains consistent.

The time required to reach a consistent state depends, on how long the application needs to recover each transaction.

Rollback In some cases you may need to “rollback” or undo a transaction when the application needs to “cancel” the transaction, or because it can never recover as in cases where one of the accounts doesn’t exist, or stops existing during the transaction.

There are two possible rollback operations:

1. After you [apply the transaction](#) (page 68) (i.e. the third step,) you have fully committed the transaction and you should not roll back the transaction. Instead, create a new transaction and switch the values in the source and destination fields.
2. After you [create the transaction](#) (page 67) (i.e. the first step,) but before you [apply the transaction](#) (page 68) (i.e. the third step,) use the following process:

Set Transaction State to Canceling Begin by setting the transaction’s state to `canceling` using the following `update()` operation:

```
db.transactions.update({_id: t._id}, {$set: {state: "canceling"})})
```

Undo the Transaction Use the following sequence of operations to undo the transaction operation from both accounts:

```
db.accounts.update({name: t.source, pendingTransactions: t._id}, {$inc: {balance: t.value}}, $pull: {  
  pendingTransactions: t._id}  
db.accounts.update({name: t.destination, pendingTransactions: t._id}, {$inc: {balance: -t.value}}, $pull: {  
  pendingTransactions: t._id}  
db.accounts.find()
```

The `find()` operation will return the contents of the `accounts` collection, which should resemble the following:

```
{ "_id" : ObjectId("4d7bc97fb8a04f5126961523"), "balance" : 1000, "name" : "A", "pendingTransactions": []  
{ "_id" : ObjectId("4d7bc984b8a04f5126961524"), "balance" : 1000, "name" : "B", "pendingTransactions": []}
```

Set Transaction State to Canceled Finally, use the following `update()` operation to set the transaction's state to `canceled`:

Step 3: set the transaction's state to “canceled”:

```
db.transactions.update({_id: t._id}, {$set: {state: "canceled"}})
```

Multiple Applications Transactions exist, in part, so that several applications can create and run operations concurrently without causing data inconsistency or conflicts. As a result, it is crucial that only one application can handle a given transaction at any point in time.

Consider the following example, with a single transaction (i.e. T1) and two applications (i.e. A1 and A2). If both applications begin processing the transaction which is still in the `initial` state (i.e. [step 1](#) (page 67)), then:

- A1 can apply the entire whole transaction before A2 starts.
- A2 will then apply T1 for the second time, because the transaction does not appear as pending in the `accounts` documents.

To handle multiple applications, create a marker in the transaction document itself to identify the application that is handling the transaction. Use `findAndModify()` method to modify the transaction:

```
t = db.transactions.findAndModify({query: {state: "initial", application: {$exists: 0}},  
                                  update: {$set: {state: "pending", application: "A1"}},  
                                  new: true})
```

When you modify and reassign the local shell variable `t`, the mongo shell will return the `t` object, which should resemble the following:

```
{  
  "_id" : ObjectId("4d7be8af2c10315c0847fc85"),  
  "application" : "A1",  
  "destination" : "B",  
  "source" : "A",  
  "state" : "pending",  
  "value" : 150  
}
```

Amend the transaction operations to ensure that only applications that match the identifier in the value of the `application` field before applying the transaction.

If the application A1 fails during transaction execution, you can use the [recovery procedures](#) (page 69), but applications should ensure that they “owns” the transaction before applying the transaction. For example to resume pending jobs, use a query that resembles the following:

```
db.transactions.find({application: "A1", state: "pending"})
```

This will (or may) return a document from the `transactions` document that resembles the following:

```
{ "_id" : ObjectId("4d7be8af2c10315c0847fc85"), "application" : "A1", "destination" : "B", "source"
```

Using Two-Phase Commits in Production Applications

The example transaction above is intentionally simple. For example, it assumes that:

- it is always possible roll back operations an account.
- account balances can hold negative values.

Production implementations would likely be more complex. Typically accounts need to information about current balance, pending credits, pending debits. Then:

- when your application *switches the transaction state to pending* (page 67) (i.e. step 2) it would also make sure that the accounts has sufficient funds for the transaction. During this update operation, the application would also modify the values of the credits and debits as well as adding the transaction as pending.
- when your application *removes the pending transaction* (page 68) (i.e. step 4) the application would apply the transaction on balance, modify the credits and debits as well as removing the transaction from the pending field., all in one update.

Because all of the changes in the above two operations occur within a single `update()` operation, these changes are all atomic.

Additionally, for most important transactions, ensure that:

- the database interface (i.e. client library or *driver*) has a reasonable *write concern* configured to ensure that operations return a response on the success or failure of a write operation.
- your `mongod` instance has *journaling* enabled to ensure that your data is always in a recoverable state, in the event of an unclean `mongod` shutdown.

2.3.9 Create Tailable Cursor

Overview

By default, MongoDB will automatically close a cursor when the client has exhausted all results in the cursor. However, for *capped collections* (page 158) you may use a *Tailable Cursor* that remains open after the client exhausts the results in the initial cursor. Tailable cursors are conceptually equivalent to the `tail` Unix command with the `-f` option (i.e. with “follow” mode.) After clients insert new additional documents into a capped collection, the tailable cursor will continue to retrieve documents.

Use tailable cursors on capped collections with high numbers of write operations for which an index would be too expensive. For instance, MongoDB *replication* (page 373) uses tailable cursors to tail the primary’s *oplog*.

Note: If your query is on an indexed field, do not use tailable cursors, but instead, use a regular cursor. Keep track of the last value of the indexed field returned by the query. To retrieve the newly added documents, query the collection again using the last value of the indexed field in the query criteria, as in the following example:

```
db.<collection>.find( { indexedField: { $gt: <lastvalue> } } )
```

Consider the following behaviors related to tailable cursors:

- Tailable cursors do not use indexes and return documents in *natural order*.

- Because tailable cursors do not use indexes, the initial scan for the query may be expensive; but, after initially exhausting the cursor, subsequent retrievals of the newly added documents are inexpensive.
- Tailable cursors may become *dead*, or invalid, if either:
 - the query returns no match.
 - the cursor returns the document at the “end” of the collection and then the application deletes those documents.

A *dead* cursor has an id of 0.

See your [driver documentation](#) (page 92) for the driver-specific method to specify the tailable cursor. For more information on the details of specifying a tailable cursor, see [MongoDB wire protocol](#)¹² documentation.

C++ Example

The `tail` function uses a tailable cursor to output the results from a query to a capped collection:

- The function handles the case of the dead cursor by having the query be inside a loop.
- To periodically check for new data, the `cursor->more()` statement is also inside a loop.

```
#include "client/dbclient.h"

using namespace mongo;

/*
 * Example of a tailable cursor.
 * The function "tails" the capped collection (ns) and output elements as they are added.
 * The function also handles the possibility of a dead cursor by tracking the field 'insertDate'.
 * New documents are added with increasing values of 'insertDate'.
 */

void tail(DBClientBase& conn, const char *ns) {

    BSONElement lastValue = minKey.firstElement();

    Query query = Query().hint( BSON( "$natural" << 1 ) );

    while ( 1 ) {
        auto_ptr<DBClientCursor> c =
            conn.query(ns, query, 0, 0,
                       QueryOption_CursorTailable | QueryOption_AwaitData );

        while ( 1 ) {
            if ( !c->more() ) {

                if ( c->isDead() ) {
                    break;
                }

                continue;
            }

            BSONObj o = c->next();
            lastValue = o["insertDate"];
            cout << o.toString() << endl;
        }
    }
}
```

¹²<http://docs.mongodb.org/meta-driver/latest/legacy/mongodb-wire-protocol>

```

        }

        query = QUERY( "insertDate" << GT << lastValue ).hint( BSON( "$natural" << 1 ) );
    }
}

```

The `tail` function performs the following actions:

- Initialize the `lastValue` variable, which tracks the last accessed value. The function will use the `lastValue` if the cursor becomes *invalid* and `tail` needs to restart the query. Use `hint()` to ensure that the query uses the `$natural` order.
- In an outer `while (1)` loop,

- Query the capped collection and return a tailable cursor that blocks for several seconds waiting for new documents

```

auto_ptr<DBClientCursor> c =
    conn.query(ns, query, 0, 0,
               QueryOption_CursorTailable | QueryOption_AwaitData );

```

- * Specify the capped collection using `ns` as an argument to the function.
- * Set the `QueryOption_CursorTailable` option to create a tailable cursor.
- * Set the `QueryOption_AwaitData` option so that the returned cursor blocks for a few seconds to wait for data.

- In an inner `while (1)` loop, read the documents from the cursor:

- * If the cursor has no more documents and is not invalid, loop the inner `while` loop to recheck for more documents.
- * If the cursor has no more documents and is dead, break the inner `while` loop.
- * If the cursor has documents:
 - output the document,
 - update the `lastValue` value,
 - and loop the inner `while (1)` loop to recheck for more documents.

- If the logic breaks out of the inner `while (1)` loop and the cursor is invalid:

- * Use the `lastValue` value to create a new query condition that matches documents added after the `lastValue`. Explicitly ensure `$natural` order with the `hint()` method:

```

query = QUERY( "insertDate" << GT << lastValue ).hint( BSON( "$natural" << 1 ) );

```

- * Loop through the outer `while (1)` loop to re-query with the new query condition and repeat.

See also:

[Detailed blog post on tailable cursor¹³](#)

¹³<http://shtylman.com/post/the-tail-of-mongodb>

2.3.10 Isolate Sequence of Operations

Overview

Write operations are atomic on the level of a single document: no single write operation can atomically affect more than one document or more than one collection.

When a single write operation modifies multiple documents, the operation as a whole is not atomic, and other operations may interleave. The modification of a single document, or record, is always atomic, even if the write operation modifies multiple sub-document *within* the single record.

No other operations are atomic; however, you can *isolate* a single write operation that affects multiple documents using the `isolation` operator.

This document describes one method of updating documents *only* if the local copy of the document reflects the current state of the document in the database. In addition the following methods provide a way to manage isolated sequences of operations:

- the `findAndModify()` provides an isolated query and modify operation.
- *Perform Two Phase Commits* (page 66)
- Create a *unique index* (page 330), to ensure that a key doesn't exist when you insert it.

Update if Current

In this pattern, you will:

- query for a document,
- modify the fields in that document
- and update the fields of a document *only if* the fields have not changed in the collection since the query.

Consider the following example in JavaScript which attempts to update the `qty` field of a document in the `products` collection:

```
1 var myCollection = db.products;
2 var myDocument = myCollection.findOne( { sku: 'abc123' } );
3
4 if (myDocument) {
5
6     var oldQty = myDocument.qty;
7
8     if (myDocument.qty < 10) {
9         myDocument.qty *= 4;
10    } else if ( myDocument.qty < 20 ) {
11        myDocument.qty *= 3;
12    } else {
13        myDocument.qty *= 2;
14    }
15
16 myCollection.update(
17     {
18         _id: myDocument._id,
19         qty: oldQty
20     },
21     {
22         $set: { qty: myDocument.qty }
23     }
```

```

24     )
25
26     var err = db.getLastErrorObj();
27
28     if ( err && err.code ) {
29         print("unexpected error updating document: " + tojson( err ));
30     } else if ( err.n == 0 ) {
31         print("No update: no matching document for { _id: " + myDocument._id + ", qty: " + oldQty + " }
32     }
33
34 }
```

Your application may require some modifications of this pattern, such as:

- Use the entire document as the query in lines 18 and 19, to generalize the operation and guarantee that the original document was not modified, rather than ensuring that a single field was not changed.
- Add a version variable to the document that applications increment upon each update operation to the documents. Use this version variable in the query expression. You must be able to ensure that *all* clients that connect to your database obey this constraint.
- Use `$set` in the update expression to modify only your fields and prevent overriding other fields.
- Use one of the methods described in [Create an Auto-Incrementing Sequence Field](#) (page 75).

2.3.11 Create an Auto-Incrementing Sequence Field

Synopsis

MongoDB reserves the `_id` field in the top level of all documents as a primary key. `_id` must be unique, and always has an index with a [unique constraint](#) (page 330). However, except for the unique constraint you can use any value for the `_id` field in your collections. This tutorial describes two methods for creating an incrementing sequence number for the `_id` field using the following:

- [A Counters Collection](#) (page 75)
- [Optimistic Loop](#) (page 77)

Warning: Generally in MongoDB, you would not use an auto-increment pattern for the `_id` field, or any field, because it does not scale for databases with large numbers of documents. Typically the default value `ObjectId` is more ideal for the `_id`.

A Counters Collection

Use a separate `counters` collection to track the *last* number sequence used. The `_id` field contains the sequence name and the `seq` field contains the last value of the sequence.

1. Insert into the `counters` collection, the initial value for the `userid`:

```

db.counters.insert(
{
    _id: "userid",
    seq: 0
})
```

2. Create a `getNextSequence` function that accepts a name of the sequence. The function uses the `findAndModify()` method to atomically increment the `seq` value and return this new value:

```
function getNextSequence(name) {
  var ret = db.counters.findAndModify(
    {
      query: { _id: name },
      update: { $inc: { seq: 1 } },
      new: true
    }
  );

  return ret.seq;
}
```

3. Use this `getNextSequence()` function during `insert()`.

```
db.users.insert(
  {
    _id: getNextSequence("userid"),
    name: "Sarah C."
  }
)

db.users.insert(
  {
    _id: getNextSequence("userid"),
    name: "Bob D."
  }
)
```

You can verify the results with `find()`:

```
db.users.find()
```

The `_id` fields contain incrementing sequence values:

```
{
  _id : 1,
  name : "Sarah C."
}
{
  _id : 2,
  name : "Bob D."
}
```

Note: When `findAndModify()` includes the `upsert: true` option **and** the query field(s) is not uniquely indexed, the method could insert a document multiple times in certain circumstances. For instance, if multiple clients each invoke the method with the same query condition and these methods complete the find phase before any of methods perform the modify phase, these methods could insert the same document.

In the `counters` collection example, the query field is the `_id` field, which always has a unique index. Consider that the `findAndModify()` includes the `upsert: true` option, as in the following modified example:

```
function getNextSequence(name) {
  var ret = db.counters.findAndModify(
    {
      query: { _id: name },
      update: { $inc: { seq: 1 } },
      new: true,
```

```

        upsert: true
    }
);

return ret.seq;
}

```

If multiple clients were to invoke the `getNextSequence()` method with the same name parameter, then the methods would observe one of the following behaviors:

- Exactly one `findAndModify()` would successfully insert a new document.
- Zero or more `findAndModify()` methods would update the newly inserted document.
- Zero or more `findAndModify()` methods would fail when they attempted to insert a duplicate.

If the method fails due to a unique index constraint violation, retry the method. Absent a delete of the document, the retry should not fail.

Optimistic Loop

In this pattern, an *Optimistic Loop* calculates the incremented `_id` value and attempts to insert a document with the calculated `_id` value. If the insert is successful, the loop ends. Otherwise, the loop will iterate through possible `_id` values until the insert is successful.

1. Create a function named `insertDocument` that performs the “insert if not present” loop. The function wraps the `insert()` method and takes a `doc` and a `targetCollection` arguments.

```

function insertDocument(doc, targetCollection) {

    while (1) {

        var cursor = targetCollection.find( {}, { _id: 1 } ).sort( { _id: -1 } ).limit(1);

        var seq = cursor.hasNext() ? cursor.next()._id + 1 : 1;

        doc._id = seq;

        targetCollection.insert(doc);

        var err = db.getLastErrorObj();

        if( err && err.code ) {
            if( err.code == 11000 /* dup key */ )
                continue;
            else
                print( "unexpected error inserting data: " + toJson( err ) );
        }

        break;
    }
}

```

The `while (1)` loop performs the following actions:

- Queries the `targetCollection` for the document with the maximum `_id` value.
- Determines the next sequence value for `_id` by:

- adding 1 to the returned `_id` value if the returned cursor points to a document.
 - otherwise: it sets the next sequence value to 1 if the returned cursor points to no document.
- For the `doc` to insert, set its `_id` field to the calculated sequence value `seq`.
 - Insert the `doc` into the `targetCollection`.
 - If the insert operation errors with duplicate key, repeat the loop. Otherwise, if the insert operation encounters some other error or if the operation succeeds, break out of the loop.

2. Use the `insertDocument()` function to perform an insert:

```
var myCollection = db.users2;
```

```
insertDocument ( { name: "Grace H." }, myCollection );  
  
insertDocument ( { name: "Ted R." }, myCollection );
```

You can verify the results with `find()`:

```
db.users2.find()
```

The `_id` fields contain incrementing sequence values:

```
{ _id: 1, name: "Grace H." }  
{ _id : 2, "name" : "Ted R." }
```

The while loop may iterate many times in collections with larger insert volumes.

2.3.12 Limit Number of Elements in an Array after an Update

New in version 2.4.

Synopsis

Consider an application where users may submit many scores (e.g. for a test), but the application only needs to track the top three test scores.

This pattern uses the `$push` operator with the `$each`, `$sort`, and `$slice` modifiers to sort and maintain an array of fixed size.

Important: The array elements must be documents in order to use the `$sort` modifier.

Pattern

Consider the following document in the collection `students`:

```
{
  _id: 1,
  scores: [
    { attempt: 1, score: 10 },
    { attempt: 2, score: 8 }
  ]
}
```

The following update uses the `$push` operator with:

- the `$each` modifier to append to the array 2 new elements,
- the `$sort` modifier to order the elements by ascending (1) score, and
- the `$slice` modifier to keep the last 3 elements of the ordered array.

```
db.students.update(
  { _id: 1 },
  { $push: { scores: { $each : [
      { attempt: 3, score: 7 },
      { attempt: 4, score: 4 }
    ],
    $sort: { score: 1 },
    $slice: -3
  } }
}
```

Note: When using the `$sort` modifier on the array element, access the field in the subdocument element directly instead of using the *dot notation* on the array field.

After the operation, the document contains the only the top 3 scores in the `scores` array:

```
{
  "_id" : 1,
  "scores" : [
    { "attempt" : 3, "score" : 7 },
    { "attempt" : 2, "score" : 8 },
    { "attempt" : 1, "score" : 10 }
  ]
}
```

See also:

- `$push` operator,
- `$each` modifier,
- `$sort` modifier, and
- `$slice` modifier.

2.4 MongoDB CRUD Reference

2.4.1 Query Cursor Methods

Name	Description
<code>cursor.count()</code>	Returns a count of the documents in a cursor.
<code>cursor.explain()</code>	Reports on the query execution plan, including index use, for a cursor.
<code>cursor.hint()</code>	Forces MongoDB to use a specific index for a query.
<code>cursor.limit()</code>	Constrains the size of a cursor's result set.
<code>cursor.next()</code>	Returns the next document in a cursor.
<code>cursor.skip()</code>	Returns a cursor that begins returning results only after passing or skipping a number of documents.
<code>cursor.sort()</code>	Returns results ordered according to a sort specification.
<code>cursor.toArray()</code>	Returns an array that contains all documents returned by the cursor.

2.4.2 Query and Data Manipulation Collection Methods

Name	Description
<code>db.collection.count()</code>	Wraps <code>count</code> to return a count of the number of documents in a collection or matching a query.
<code>db.collection.distinct()</code>	Returns an array of documents that have distinct values for the specified field.
<code>db.collection.find()</code>	Performs a query on a collection and returns a cursor object.
<code>db.collection.findOne()</code>	Performs a query and returns a single document.
<code>db.collection.insert()</code>	Creates a new document in a collection.
<code>db.collection.remove()</code>	Deletes documents from a collection.
<code>db.collection.save()</code>	Provides a wrapper around an <code>insert()</code> and <code>update()</code> to insert new documents.
<code>db.collection.update()</code>	Modifies a document in a collection.

2.4.3 MongoDB CRUD Reference Documentation

[Write Concern Reference](#) (page 80) Configuration options associated with the guarantee MongoDB provides when reporting on the success of a write operation.

[SQL to MongoDB Mapping Chart](#) (page 82) An overview of common database operations showing both the MongoDB operations and SQL statements.

[The bios Example Collection](#) (page 87) Sample data for experimenting with MongoDB. `insert()`, `update()` and `find()` pages use the data for some of their examples.

[MongoDB Drivers and Client Libraries](#) (page 92) Applications access MongoDB using client libraries, or drivers, that provide idiomatic interfaces to MongoDB for many programming languages and development environments.

Write Concern Reference

Overview

Write concern describes the guarantee that MongoDB provides when reporting on the success of a write operation. The strength of the write concerns determine the level of guarantee. When inserts, updates and deletes have a *weak* write concern, write operations return quickly. In some failure cases, write operations issued with weak write concerns

may not persist. With *stronger* write concerns, clients wait after sending a write operation for MongoDB to confirm the write operations.

MongoDB provides different levels of write concern to better address the specific needs of applications. Clients may adjust write concern to ensure that the most important operations persist successfully to an entire MongoDB deployment. For other less critical operations, clients can adjust the write concern to ensure faster performance rather than ensure persistence to the entire deployment.

See also:

[Write Concern](#) (page 44) for an introduction to write concern in MongoDB.

Available Write Concern

To provide write concern, [drivers](#) (page 92) issue the `getLastError` command after a write operation and receive a document with information about the last operation. This document’s `err` field contains either:

- `null`, which indicates the write operations have completed successfully, or
- a description of the last error encountered.

The definition of a “successful write” depends on the arguments specified to `getLastError`, or in replica sets, the configuration of [getLastErrorDefaults](#) (page 477). When deciding the level of write concern for your application, see the introduction to [Write Concern](#) (page 44).

The `getLastError` command has the following options to configure write concern requirements:

- `j` or “journal” option

This option confirms that the `mongod` instance has written the data to the on-disk journal and ensures data is not lost if the `mongod` instance shuts down unexpectedly. Set to `true` to enable, as shown in the following example:

```
db.runCommand( { getLastError: 1, j: "true" } )
```

If you set `journal` to `true`, and the `mongod` does not have journaling enabled, as with `nojournal`, then `getLastError` will provide basic receipt acknowledgment, and will include a `jnote` field in its return document.

- `w` option

This option provides the ability to disable write concern entirely *as well as* specifies the write concern operations for *replica sets*. See [Write Concern Considerations](#) (page 44) for an introduction to the fundamental concepts of write concern. By default, the `w` option is set to `1`, which provides basic receipt acknowledgment on a single `mongod` instance or on the *primary* in a replica set.

The `w` option takes the following values:

– `-1`:

Disables all acknowledgment of write operations, and suppresses all errors, including network and socket errors.

– `0`:

Disables basic acknowledgment of write operations, but returns information about socket exceptions and networking errors to the application.

Note: If you disable basic write operation acknowledgment but require journal commit acknowledgment, the journal commit prevails, and the driver will require that `mongod` will acknowledge the write operation.

- 1:

Provides acknowledgment of write operations on a standalone `mongod` or the *primary* in a replica set.

- *A number greater than 1:*

Guarantees that write operations have propagated successfully to the specified number of replica set members including the primary. If you set `w` to a number that is greater than the number of set members that hold data, MongoDB waits for the non-existent members to become available, which means MongoDB blocks indefinitely.

- *majority:*

Confirms that write operations have propagated to the majority of configured replica set: a majority of the set's configured members must acknowledge the write operation before it succeeds. This ensures that write operation will *never* be subject to a rollback in the course of normal operation, and furthermore allows you to avoid hard coding assumptions about the size of your replica set into your application.

- *A tag set:*

By specifying a [tag set](#) (page 446) you can have fine-grained control over which replica set members must acknowledge a write operation to satisfy the required level of write concern.

`getLastError` also supports a `wtimeout` setting which allows clients to specify a timeout for the write concern: if you don't specify `wtimeout`, or if you give it a value of 0, and the `mongod` cannot fulfill the write concern the `getLastError` will block, potentially forever.

For more information on write concern and replica sets, see [Write Concern for Replica Sets](#) (page 47) for more information.

In sharded clusters, `mongos` instances will pass write concern on to the shard `mongod` instances.

SQL to MongoDB Mapping Chart

In addition to the charts that follow, you might want to consider the [Frequently Asked Questions](#) (page 571) section for a selection of common questions about MongoDB.

Terminology and Concepts

The following table presents the various SQL terminology and concepts and the corresponding MongoDB terminology and concepts.

SQL Terms/Concepts	MongoDB Terms/Concepts
<code>database</code>	<i>database</i>
<code>table</code>	<i>collection</i>
<code>row</code>	<i>document</i> or <i>JSON</i> document
<code>column</code>	<i>field</i>
<code>index</code>	<i>index</i>
<code>table joins</code>	embedded documents and linking
<code>primary key</code>	<i>primary key</i>
Specify any unique column or column combination as primary key.	In MongoDB, the primary key is automatically set to the <code>_id</code> field.
aggregation (e.g. group by)	aggregation pipeline See the SQL to Aggregation Mapping Chart (page 306).

Executables

The following table presents the MySQL/Oracle executables and the corresponding MongoDB executables.

	MySQL/Oracle	MongoDB
Database Server	mysqld/oracle	mongod
Database Client	mysql/sqlplus	mongo

Examples

The following table presents the various SQL statements and the corresponding MongoDB statements. The examples in the table assume the following conditions:

- The SQL examples assume a table named `users`.
- The MongoDB examples assume a collection named `users` that contain documents of the following prototype:

```
{  
    _id: ObjectId("509a8fb2f3f4948bd2f983a0"),  
    user_id: "abc123",  
    age: 55,  
    status: 'A'  
}
```

Create and Alter The following table presents the various SQL statements related to table-level actions and the corresponding MongoDB statements.

SQL Schema Statements	MongoDB Schema Statements	Reference
<pre>CREATE TABLE users (id MEDIUMINT NOT NULL AUTO_INCREMENT, user_id Varchar(30), age Number, status char(1), PRIMARY KEY (id))</pre>	<p>Implicitly created on first <code>insert()</code> operation. The primary key <code>_id</code> is automatically added if <code>_id</code> field is not specified.</p> <pre>db.users.insert({ user_id: "abc123", age: 55, status: "A" })</pre> <p>However, you can also explicitly create a collection:</p> <pre>db.createCollection("users")</pre>	See <code>insert()</code> and <code>db.createCollection()</code> for more information.
<pre>ALTER TABLE users ADD join_date DATETIME</pre>	<p>Collections do not describe or enforce the structure of its documents; i.e. there is no structural alteration at the collection level.</p> <p>However, at the document level, <code>update()</code> operations can add fields to existing documents using the <code>\$set</code> operator.</p> <pre>db.users.update({ }, { \$set: { join_date: new Date() } }, { multi: true })</pre>	See the Data Modeling Concepts (page 97), <code>update()</code> , and <code>\$set</code> for more information on changing the structure of documents in a collection.
<pre>ALTER TABLE users DROP COLUMN join_date</pre>	<p>Collections do not describe or enforce the structure of its documents; i.e. there is no structural alteration at the collection level.</p> <p>However, at the document level, <code>update()</code> operations can remove fields from documents using the <code>\$unset</code> operator.</p> <pre>db.users.update({ }, { \$unset: { join_date: "" } }, { multi: true })</pre>	See Data Modeling Concepts (page 97), <code>update()</code> , and <code>\$unset</code> for more information on changing the structure of documents in a collection.
<pre>CREATE INDEX idx_user_id_as ON users(user_id)</pre>	<pre>db.users.ensureIndex({ user_id: 1 })</pre>	See <code>ensureIndex()</code> and indexes (page 314) for more information.
<pre>CREATE INDEX idx_user_id_asc_age_desc ON users(user_id, age DESC)</pre>	<pre>db.users.ensureIndex({ user_id: 1, age: -1 })</pre>	See <code>ensureIndex()</code> and indexes (page 314) for more information.
<pre>DROP TABLE users</pre>	<pre>db.users.drop()</pre>	See <code>drop()</code> for more information.

Insert The following table presents the various SQL statements related to inserting records into tables and the corresponding MongoDB statements.

SQL INSERT Statements	MongoDB insert() Statements	Reference
<pre>INSERT INTO users(user_id, age, status) VALUES ("bcd001", 45, "A")</pre>	<pre>db.users.insert({ user_id: "bcd001", age: 45, status: "A" })</pre>	See <code>insert()</code> for more information.

Select The following table presents the various SQL statements related to reading records from tables and the corresponding MongoDB statements.

SQL SELECT Statements	MongoDB find() Statements	Reference
<code>SELECT * FROM users</code>	<code>db.users.find()</code>	See find() for more information.
<code>SELECT id, user_id, status FROM users</code>	<code>db.users.find({ }, { user_id: 1, status: 1 })</code>	See find() for more information.
<code>SELECT user_id, status FROM users</code>	<code>db.users.find({ }, { user_id: 1, status: 1, _id: 0 })</code>	See find() for more information.
<code>SELECT * FROM users WHERE status = "A"</code>	<code>db.users.find({ status: "A" })</code>	See find() for more information.
<code>SELECT user_id, status FROM users WHERE status = "A"</code>	<code>db.users.find({ status: "A" }, { user_id: 1, status: 1, _id: 0 })</code>	See find() for more information.
<code>SELECT * FROM users WHERE status != "A"</code>	<code>db.users.find({ status: { \$ne: "A" } })</code>	See find() and \$ne for more information.
<code>SELECT * FROM users WHERE status = "A" AND age = 50</code>	<code>db.users.find({ status: "A", age: 50 })</code>	See find() and \$and for more information.
<code>SELECT * FROM users WHERE status = "A" OR age = 50</code>	<code>db.users.find({ \$or: [{ status: "A" }, { age: 50 }] })</code>	See find() and \$or for more information.
<code>SELECT * FROM users WHERE age > 25</code>	<code>db.users.find({ age: { \$gt: 25 } })</code>	See find() and \$gt for more information.
<code>SELECT * FROM users WHERE age < 25</code>	<code>db.users.find({ age: { \$lt: 25 } })</code>	See find() and \$lt for more information.
<code>SELECT * FROM users WHERE age > 25 AND age <= 50</code>	<code>db.users.find({ age: { \$gt: 25, \$lte: 50 } })</code>	See find() , \$gt , and \$lte for more information.
86 <code>SELECT * FROM users WHERE user_id like "%bc%"</code>	<code>db.users.find({ user_id: /bc/ })</code>	Chapter 2. MongoDB CRUD Operations See find() for more information.

Update Records The following table presents the various SQL statements related to updating existing records in tables and the corresponding MongoDB statements.

SQL Update Statements	MongoDB update() Statements	Reference
<pre>UPDATE users SET status = "C" WHERE age > 25</pre>	<pre>db.users.update({ age: { \$gt: 25 } }, { \$set: { status: "C" } } { multi: true })</pre>	See <code>update()</code> , <code>\$gt</code> , and <code>\$set</code> for more information. },
<pre>UPDATE users SET age = age + 3 WHERE status = "A"</pre>	<pre>db.users.update({ status: "A" } , { \$inc: { age: 3 } }, { multi: true })</pre>	See <code>update()</code> , <code>\$inc</code> , and <code>\$set</code> for more information.

Delete Records The following table presents the various SQL statements related to deleting records from tables and the corresponding MongoDB statements.

SQL Delete Statements	MongoDB remove() Statements	Reference
<pre>DELETE FROM users WHERE status = "D"</pre>	<pre>db.users.remove({ status: "D" })</pre>	See <code>remove()</code> for more information.
<pre>DELETE FROM users</pre>	<pre>db.users.remove()</pre>	See <code>remove()</code> for more information.

The bios Example Collection

The `bios` collection provides example data for experimenting with MongoDB. Many of this guide's examples on insert, update and read operations create or query data from the `bios` collection.

The following documents comprise the `bios` collection. In the examples, the data might be different, as the examples themselves make changes to the data.

```
{
  "_id" : 1,
  "name" : {
    "first" : "John",
    "last" : "Backus"
  },
  "birth" : ISODate("1924-12-03T05:00:00Z"),
  "death" : ISODate("2007-03-17T04:00:00Z"),
  "contribs" : [
    "Fortran",
    "ALGOL",
    "Backus-Naur Form",
    "FP"
  ],
  "awards" : [
    {
      "award" : "W.W. McDowell Award",
      "year" : 1967,
      "by" : "IEEE Computer Society"
    }
  ]
}
```

```
        },
        {
            "award" : "National Medal of Science",
            "year" : 1975,
            "by" : "National Science Foundation"
        },
        {
            "award" : "Turing Award",
            "year" : 1977,
            "by" : "ACM"
        },
        {
            "award" : "Draper Prize",
            "year" : 1993,
            "by" : "National Academy of Engineering"
        }
    ]
}
{
    "_id" : ObjectId("51df07b094c6acd67e492f41"),
    "name" : {
        "first" : "John",
        "last" : "McCarthy"
    },
    "birth" : ISODate("1927-09-04T04:00:00Z"),
    "death" : ISODate("2011-12-24T05:00:00Z"),
    "contribs" : [
        "Lisp",
        "Artificial Intelligence",
        "ALGOL"
    ],
    "awards" : [
        {
            "award" : "Turing Award",
            "year" : 1971,
            "by" : "ACM"
        },
        {
            "award" : "Kyoto Prize",
            "year" : 1988,
            "by" : "Inamori Foundation"
        },
        {
            "award" : "National Medal of Science",
            "year" : 1990,
            "by" : "National Science Foundation"
        }
    ]
}
{
    "_id" : 3,
    "name" : {
        "first" : "Grace",
        "last" : "Hopper"
    },
    "title" : "Rear Admiral",
```

```
"birth" : ISODate("1906-12-09T05:00:00Z"),
"death" : ISODate("1992-01-01T05:00:00Z"),
"contribs" : [
    "UNIVAC",
    "compiler",
    "FLOW-MATIC",
    "COBOL"
],
"awards" : [
    {
        "award" : "Computer Sciences Man of the Year",
        "year" : 1969,
        "by" : "Data Processing Management Association"
    },
    {
        "award" : "Distinguished Fellow",
        "year" : 1973,
        "by" : " British Computer Society"
    },
    {
        "award" : "W. W. McDowell Award",
        "year" : 1976,
        "by" : "IEEE Computer Society"
    },
    {
        "award" : "National Medal of Technology",
        "year" : 1991,
        "by" : "United States"
    }
]
},
{
    "_id" : 4,
    "name" : {
        "first" : "Kristen",
        "last" : "Nygaard"
    },
    "birth" : ISODate("1926-08-27T04:00:00Z"),
    "death" : ISODate("2002-08-10T04:00:00Z"),
    "contribs" : [
        "OOP",
        "Simula"
    ],
    "awards" : [
        {
            "award" : "Rosing Prize",
            "year" : 1999,
            "by" : "Norwegian Data Association"
        },
        {
            "award" : "Turing Award",
            "year" : 2001,
            "by" : "ACM"
        },
        {
            "award" : "IEEE John von Neumann Medal",
            "year" : 2001,
            "by" : "IEEE"
        }
    ]
}
```

```
        "by" : "IEEE"
    }
]
}

{
    "_id" : 5,
    "name" : {
        "first" : "Ole-Johan",
        "last" : "Dahl"
    },
    "birth" : ISODate("1931-10-12T04:00:00Z"),
    "death" : ISODate("2002-06-29T04:00:00Z"),
    "contribs" : [
        "OOP",
        "Simula"
    ],
    "awards" : [
        {
            "award" : "Rosing Prize",
            "year" : 1999,
            "by" : "Norwegian Data Association"
        },
        {
            "award" : "Turing Award",
            "year" : 2001,
            "by" : "ACM"
        },
        {
            "award" : "IEEE John von Neumann Medal",
            "year" : 2001,
            "by" : "IEEE"
        }
    ]
}

{
    "_id" : 6,
    "name" : {
        "first" : "Guido",
        "last" : "van Rossum"
    },
    "birth" : ISODate("1956-01-31T05:00:00Z"),
    "contribs" : [
        "Python"
    ],
    "awards" : [
        {
            "award" : "Award for the Advancement of Free Software",
            "year" : 2001,
            "by" : "Free Software Foundation"
        },
        {
            "award" : "NLUUG Award",
            "year" : 2003,
            "by" : "NLUUG"
        }
    ]
}
```

```

}

{
    "_id" : ObjectId("51e062189c6ae665454e301d"),
    "name" : {
        "first" : "Dennis",
        "last" : "Ritchie"
    },
    "birth" : ISODate("1941-09-09T04:00:00Z"),
    "death" : ISODate("2011-10-12T04:00:00Z"),
    "contribs" : [
        "UNIX",
        "C"
    ],
    "awards" : [
        {
            "award" : "Turing Award",
            "year" : 1983,
            "by" : "ACM"
        },
        {
            "award" : "National Medal of Technology",
            "year" : 1998,
            "by" : "United States"
        },
        {
            "award" : "Japan Prize",
            "year" : 2011,
            "by" : "The Japan Prize Foundation"
        }
    ]
}

{
    "_id" : 8,
    "name" : {
        "first" : "Yukihiro",
        "aka" : "Matz",
        "last" : "Matsumoto"
    },
    "birth" : ISODate("1965-04-14T04:00:00Z"),
    "contribs" : [
        "Ruby"
    ],
    "awards" : [
        {
            "award" : "Award for the Advancement of Free Software",
            "year" : "2011",
            "by" : "Free Software Foundation"
        }
    ]
}

{
    "_id" : 9,
    "name" : {
        "first" : "James",
        "last" : "Gosling"
}

```

```
},
"birth" : ISODate("1955-05-19T04:00:00Z"),
"contribs" : [
    "Java"
],
"awards" : [
    {
        "award" : "The Economist Innovation Award",
        "year" : 2002,
        "by" : "The Economist"
    },
    {
        "award" : "Officer of the Order of Canada",
        "year" : 2007,
        "by" : "Canada"
    }
]
}

{
    "_id" : 10,
    "name" : {
        "first" : "Martin",
        "last" : "Odersky"
    },
    "contribs" : [
        "Scala"
    ],
}
```

MongoDB Drivers and Client Libraries

An application communicates with MongoDB by way of a client library, called a [driver¹⁴](#), that handles all interaction with the database in a language appropriate to the application.

Drivers

See the following pages for more information about the MongoDB drivers¹⁵:

- JavaScript ([Language Center¹⁶](#), [docs¹⁷](#))
- Python ([Language Center¹⁸](#), [docs¹⁹](#))
- Ruby ([Language Center²⁰](#), [docs²¹](#))
- PHP ([Language Center²²](#), [docs²³](#))

¹⁴<http://docs.mongodb.org/ecosystem/drivers>

¹⁵<http://docs.mongodb.org/ecosystem/drivers>

¹⁶<http://docs.mongodb.org/ecosystem/drivers/javascript>

¹⁷<http://api.mongodb.org/js/current>

¹⁸<http://docs.mongodb.org/ecosystem/drivers/python>

¹⁹<http://api.mongodb.org/python/current>

²⁰<http://docs.mongodb.org/ecosystem/drivers/ruby>

²¹<http://api.mongodb.org/ruby/current>

²²<http://docs.mongodb.org/ecosystem/drivers/php>

²³<http://php.net/mongo/>

- Perl ([Language Center²⁴](#), [docs²⁵](#))
- Java ([Language Center²⁶](#), [docs²⁷](#))
- Scala ([Language Center²⁸](#), [docs²⁹](#))
- C# ([Language Center³⁰](#), [docs³¹](#))
- C ([Language Center³²](#), [docs³³](#))
- C++ ([Language Center³⁴](#), [docs³⁵](#))
- Haskell ([Language Center³⁶](#), [docs³⁷](#))
- Erlang ([Language Center³⁸](#), [docs³⁹](#))

Driver Version Numbers

Driver version numbers use [semantic versioning⁴⁰](#) or “**major.minor.patch**” versioning system. The first number is the major version, the second the minor version, and the third indicates a patch.

Example

Driver version numbers.

If your driver has a version number of `2.9.1`, 2 is the major version, 9 is minor, and 1 is the patch.

The numbering scheme for drivers differs from the scheme for the MongoDB server. For more information on server versioning, see [MongoDB Version Numbers](#) (page 655).

²⁴<http://docs.mongodb.org/ecosystem/drivers/perl>

²⁵<http://api.mongodb.org/perl/current/>

²⁶<http://docs.mongodb.org/ecosystem/drivers/java>

²⁷<http://api.mongodb.org/java/current>

²⁸<http://docs.mongodb.org/ecosystem/drivers/scala>

²⁹<http://api.mongodb.org/scala/casbah/current/>

³⁰<http://docs.mongodb.org/ecosystem/drivers/csharp>

³¹<http://api.mongodb.org/csharp/current/>

³²<http://docs.mongodb.org/ecosystem/drivers/c>

³³<http://api.mongodb.org/c/current/>

³⁴<http://docs.mongodb.org/ecosystem/drivers/cpp>

³⁵<http://api.mongodb.org/cplusplus/current/>

³⁶<http://hackage.haskell.org/package/mongoDB>

³⁷<http://api.mongodb.org/haskell/mongodb>

³⁸<http://docs.mongodb.org/ecosystem/drivers/erlang>

³⁹<http://api.mongodb.org/erlang/mongodbs>

⁴⁰<http://semver.org/>

Data Models

Data in MongoDB has a *flexible schema*. Collections do not enforce *document* structure. This flexibility gives you data-modeling choices to match your application and its performance requirements.

Read the [Data Modeling Introduction](#) (page 95) document for a high level introduction to data modeling, and proceed to the documents in the [Data Modeling Concepts](#) (page 97) section for additional documentation of the data model design process. The [Data Model Examples and Patterns](#) (page 104) documents provide examples of different data models. In addition, the MongoDB Use Case Studies¹ provide overviews of application design and include example data models with MongoDB.

[Data Modeling Introduction](#) (page 95) An introduction to data modeling in MongoDB.

[Data Modeling Concepts](#) (page 97) The core documentation detailing the decisions you must make when determining a data model, and discussing considerations that should be taken into account.

[Data Model Examples and Patterns](#) (page 104) Examples of possible data models that you can use to structure your MongoDB documents.

[Data Model Reference](#) (page 119) Reference material for data modeling for developers of MongoDB applications.

3.1 Data Modeling Introduction

Data in MongoDB has a *flexible schema*. Unlike SQL databases, where you must determine and declare a table's schema before inserting data, MongoDB's *collections* do not enforce *document* structure. This flexibility facilitates the mapping of documents to an entity or an object. Each document can match the data fields of the represented entity, even if the data has substantial variation. In practice, however, the documents in a collection share a similar structure.

The key challenge in data modeling is balancing the needs of the application, the performance characteristics of database engine, and the data retrieval patterns. When designing data models, always consider the application usage of the data (i.e. queries, updates, and processing of the data) as well as the inherent structure of the data itself.

3.1.1 Document Structure

The key decision in designing data models for MongoDB applications revolves around the structure of documents and how the application represents relationships between data. There are two tools that allow applications to represent these relationships: *references* and *embedded documents*.

¹<http://docs.mongodb.org/ecosystem/use-cases>

References

References store capture the relationships between data by including links or *references* from one document to another. Applications can resolve these [references](#) (page 122) to access the related data. Broadly, these are *normalized* data models.

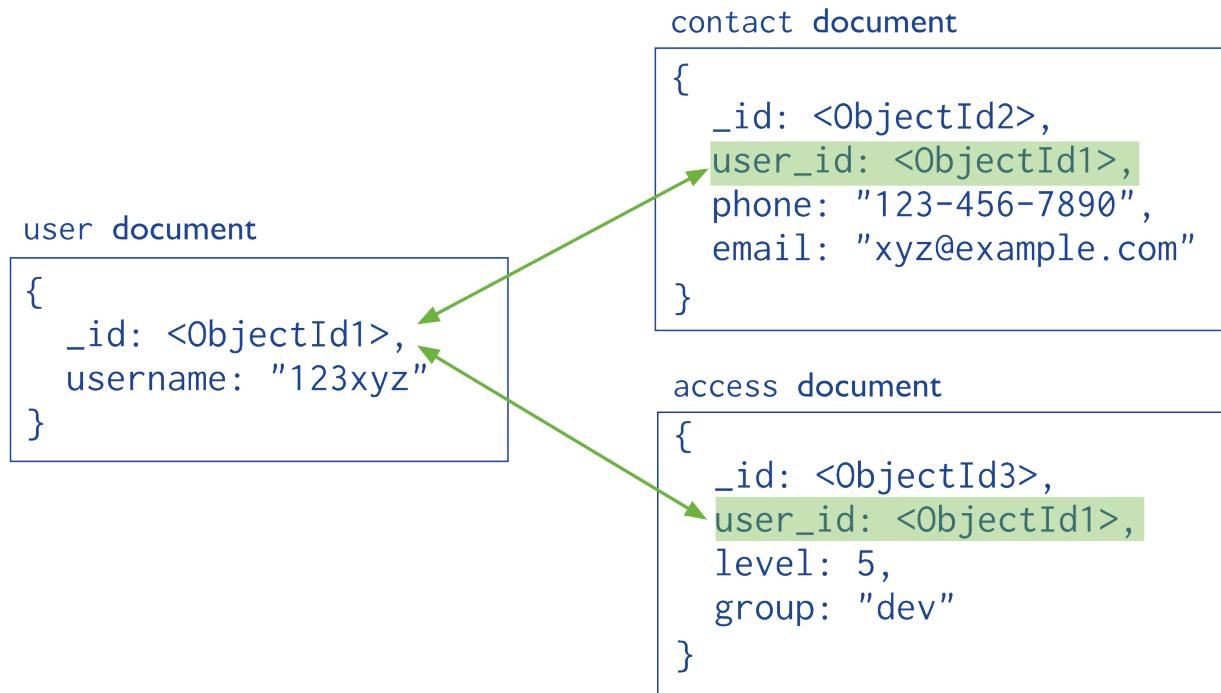


Figure 3.1: Data model using references to link documents. Both the contact document and the access document contain a reference to the user document.

See [Normalized Data Models](#) (page 99) for the strengths and weaknesses of using references.

Embedded Data

Embedded documents capture relationships between data by storing related data in a single document structure. MongoDB documents make it possible to embed document structures as sub-documents in a field or array within a document. These *denormalized* data models allow applications to retrieve and manipulate related data in a single database operation.

See [Embedded Data Models](#) (page 98) for the strengths and weaknesses of embedding sub-documents.

3.1.2 Atomicity of Write Operations

In MongoDB, write operations are atomic at the *document* level, and no single write operation can atomically affect more than one document or more than one collection. A denormalized data model with embedded data combines all related data for a represented entity in a single document. This facilitates atomic write operations since a single write operation can insert or update the data for an entity. Normalizing the data would split the data across multiple collections and would require multiple write operations that are not atomic collectively.



Figure 3.2: Data model with embedded fields that contain all related information.

However, schemas that facilitate atomic writes may limit ways that applications can use the data or may limit ways to modify applications. The [Atomicity Considerations](#) (page 100) documentation describes the challenge of designing a schema that balances flexibility and atomicity.

3.1.3 Document Growth

Some updates, such as pushing elements to an array or adding new fields, increase a *document*'s size. If the document size exceeds the allocated space for that document, MongoDB relocates the document on disk. The growth consideration can affect the decision to normalize or denormalize data. See [Document Growth Considerations](#) (page 100) for more about planning for and managing document growth in MongoDB.

3.1.4 Data Use and Performance

When designing a data model, consider how applications will use your database. For instance, if your application only uses recently inserted documents, consider using [Capped Collections](#) (page 158). Or if your application needs are mainly read operations to a collection, adding indexes to support common queries can improve performance.

See [Operational Factors and Data Models](#) (page 100) for more information on these and other operational considerations that affect data model designs.

3.2 Data Modeling Concepts

When constructing a data model for your MongoDB collection, there are various options you can choose from, each of which has its strengths and weaknesses. The following sections guide you through key design decisions and detail various considerations for choosing the best data model for your application needs.

For a general introduction to data modeling in MongoDB, see the [Data Modeling Introduction](#) (page 95). For example data models, see [Data Modeling Examples and Patterns](#) (page 104).

Data Model Design (page 98) Presents the different strategies that you can choose from when determining your data model, their strengths and their weaknesses.

Operational Factors and Data Models (page 100) Details features you should keep in mind when designing your data model, such as lifecycle management, indexing, horizontal scalability, and document growth.

GridFS (page 102) GridFS is a specification for storing documents that exceeds the *BSON*-document size limit of 16MB.

3.2.1 Data Model Design

Effective data models support your application needs. The key consideration for the structure of your documents is the decision to *embed* (page 98) or to *use references* (page 99).

Embedded Data Models

With MongoDB, you may embed related data in a single structure or document. These schema are generally known as “denormalized” models, and take advantage of MongoDB’s rich documents. Consider the following diagram:



Figure 3.3: Data model with embedded fields that contain all related information.

Embedded data models allow applications to store related pieces of information in the same database record. As a result, applications may need to issue fewer queries and updates to complete common operations.

In general, use embedded data models when:

- you have “contains” relationships between entities. See [Model One-to-One Relationships with Embedded Documents](#) (page 104).

- you have one-to-many relationships between entities. In these relationships the “many” or child documents always appear with or are viewed in the context of the “one” or parent documents. See [Model One-to-Many Relationships with Embedded Documents](#) (page 105).

In general, embedding provides better performance for read operations, as well as the ability to request and retrieve related data in a single database operation. Embedded data models make it possible to update related data in a single atomic write operation.

However, embedding related data in documents may lead to situations where documents grow after creation. Document growth can impact write performance and lead to data fragmentation. See [Document Growth](#) (page 100) for details. Furthermore, documents in MongoDB must be smaller than the maximum BSON document size. For bulk binary data, consider [GridFS](#) (page 102).

To interact with embedded documents, use *dot notation* to “reach into” embedded documents. See [query for data in arrays](#) (page 59) and [query data in sub-documents](#) (page 59) for more examples on accessing data in arrays and embedded documents.

Normalized Data Models

Normalized data models describe relationships using [references](#) (page 122) between documents.

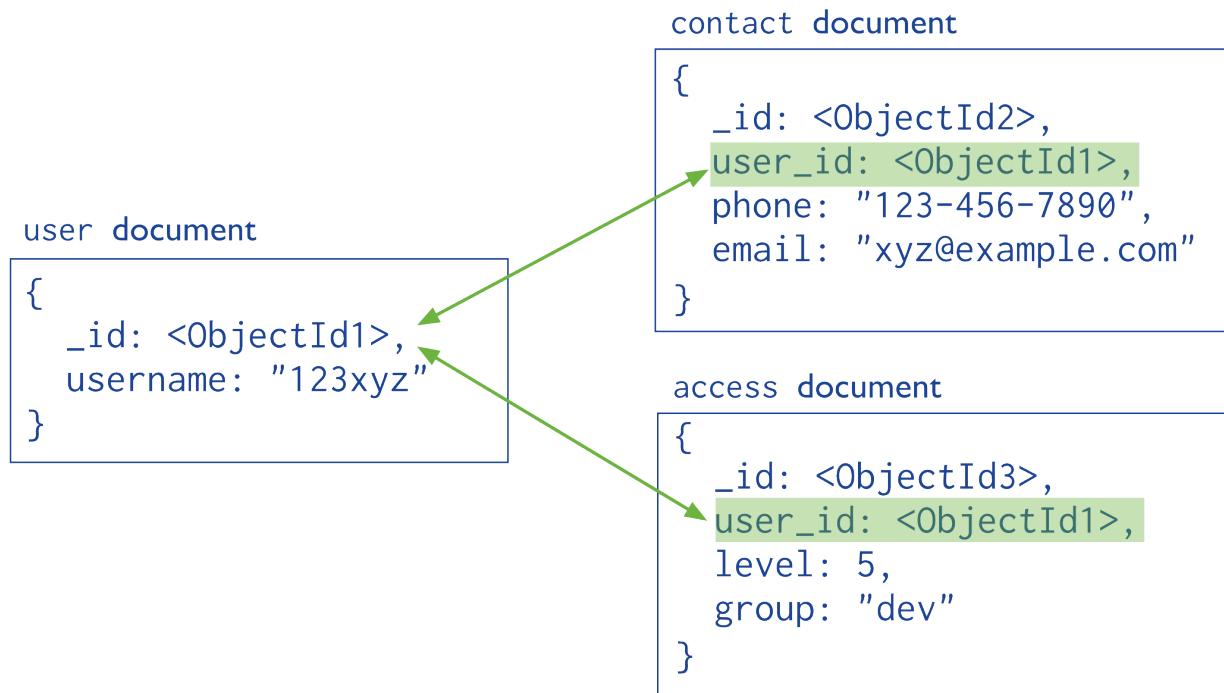


Figure 3.4: Data model using references to link documents. Both the **contact document** and the **access document** contain a reference to the **user document**.

In general, use normalized data models:

- when embedding would result in duplication of data but would not provide sufficient read performance advantages to outweigh the implications of the duplication.
- to represent more complex many-to-many relationships.

- to model large hierarchical data sets.

References provides more flexibility than embedding. However, client-side applications must issue follow-up queries to resolve the references. In other words, normalized data models can require more roundtrips to the server.

See [Model One-to-Many Relationships with Document References](#) (page 106) for an example of referencing. For examples of various tree models using references, see [Model Tree Structures](#) (page 108).

3.2.2 Operational Factors and Data Models

Modeling application data for MongoDB depends on both the data itself, as well as the characteristics of MongoDB itself. For example, different data models may allow applications to use more efficient queries, increase the throughput of insert and update operations, or distribute activity to a sharded cluster more effectively.

These factors are *operational* or address requirements that arise outside of the application but impact the performance of MongoDB based applications. When developing a data model, analyze all of your application's [read operations](#) (page 29) and [write operations](#) (page 40) in conjunction with the following considerations.

Document Growth

Some updates to documents can increase the size of documents size. These updates include, pushing elements to an array (i.e. `$push`) and adding new fields to a document. If the document size exceeds the allocated space for that document, MongoDB will relocate the document on disk. Relocating documents takes longer than *in place updates* and can lead to fragmented storage. Although MongoDB automatically [adds padding to document allocations](#) (page 54) to minimize the likelihood of relocation, data models should avoid document growth when possible.

For instance, if your applications require updates that will cause document growth, you may want to refactor your data model to use references between data in distinct documents rather than a denormalized data model.

MongoDB adaptively adjusts the amount of automatic padding to reduce occurrences of relocation. You may also use a *pre-allocation* strategy to explicitly avoid document growth. Refer to [Pre-Aggregated Reports Use Case²](#) for an example of the *pre-allocation* approach to handling document growth.

Atomicity

In MongoDB, operations are atomic at the *document* level: no **single** write operation can change more than one document or more than one collection.³ Ensure that your application stores all fields with atomic dependency requirements in the same document. If the application can tolerate non-atomic updates for two pieces of data, you can store these data in separate documents.

A data model that embeds related data in a single document facilitates these kinds of atomic operations. For data models that store references between related pieces of data, the application must issue separate read and write operations to retrieve and modify these related pieces of data.

See [Model Data for Atomic Operations](#) (page 117) for an example data model that provides atomic updates for a single document.

Sharding

MongoDB uses *sharding* to provide horizontal scaling. These clusters support deployments with large data sets and high-throughput operations. Sharding allows users to *partition* a *collection* within a database to distribute the collection's documents across a number of `mongod` instances or *shards*.

²<http://docs.mongodb.org/ecosystem/use-cases/pre-aggregated-reports>

³ Document-level atomic operations include all operations within a single MongoDB document record: operations that affect multiple sub-documents within that single record are still atomic.

To distribute data and application traffic in a sharded collection, MongoDB uses the [shard key](#) (page 499). Selecting the proper [shard key](#) (page 499) has significant implications for performance, and can enable or prevent query isolation and increased write capacity. It is important to consider carefully the field or fields to use as the shard key.

See [Sharding Introduction](#) (page 487) and [Shard Keys](#) (page 499) for more information.

Indexes

Use indexes to improve performance for common queries. Build indexes on fields that appear often in queries and for all operations that return sorted results. MongoDB automatically creates a unique index on the `_id` field.

As you create indexes, consider the following behaviors of indexes:

- Each index requires at least 8KB of data space.
- Adding an index has some negative performance impact for write operations. For collections with high write-to-read ratio, indexes are expensive since each insert must also update any indexes.
- Collections with high read-to-write ratio often benefit from additional indexes. Indexes do not affect un-indexed read operations.
- When active, each index consumes disk space and memory. This usage can be significant and should be tracked for capacity planning, especially for concerns over working set size.

See [Indexing Strategies](#) (page 363) for more information on indexes as well as [Analyze Query Performance](#) (page 63). Additionally, the MongoDB [database profiler](#) (page 171) may help identify inefficient queries.

Large Number of Collections

In certain situations, you might choose to store related information in several collections rather than in a single collection.

Consider a sample collection `logs` that stores log documents for various environment and applications. The `logs` collection contains documents of the following form:

```
{ log: "dev", ts: ..., info: ... }
{ log: "debug", ts: ..., info: ... }
```

If the total number of documents is low, you may group documents into collection by type. For `logs`, consider maintaining distinct log collections, such as `logs.dev` and `logs.debug`. The `logs.dev` collection would contain only the documents related to the dev environment.

Generally, having large number of collections has no significant performance penalty and results in very good performance. Distinct collections are very important for high-throughput batch processing.

When using models that have a large number of collections, consider the following behaviors:

- Each collection has a certain minimum overhead of a few kilobytes.
- Each index, including the index on `_id`, requires at least 8KB of data space.
- For each *database*, a single namespace file (i.e. `<database>.ns`) stores all meta-data for that database, and each index and collection has its own entry in the namespace file. MongoDB places limits on the size of namespace files.
- MongoDB has limits on the number of namespaces. You may wish to know the current number of namespaces in order to determine how many additional namespaces the database can support. To get the current number of namespaces, run the following in the `mongo` shell:

```
db.system.namespaces.count()
```

The limit on the number of namespaces depend on the `<database>.ns` size. The namespace file defaults to 16 MB.

To change the size of the *new* namespace file, start the server with the option `--nssize <new size MB>`. For existing databases, after starting up the server with `--nssize`, run the `db.repairDatabase()` command from the `mongo` shell. For impacts and considerations on running `db.repairDatabase()`, see `repairDatabase`.

Data Lifecycle Management

Data modeling decisions should take data lifecycle management into consideration.

The [Time to Live or TTL feature](#) (page 160) of collections expires documents after a period of time. Consider using the TTL feature if your application requires some data to persist in the database for a limited period of time.

Additionally, if your application only uses recently inserted documents, consider [Capped Collections](#) (page 158). Capped collections provide *first-in-first-out* (FIFO) management of inserted documents and efficiently support operations that insert and read documents based on insertion order.

3.2.3 GridFS

GridFS is a specification for storing and retrieving files that exceed the *BSON-document size limit* of 16MB.

Instead of storing a file in a single document, GridFS divides a file into parts, or chunks,⁴ and stores each of those chunks as a separate document. By default GridFS limits chunk size to 256k. GridFS uses two collections to store files. One collection stores the file chunks, and the other stores file metadata.

When you query a GridFS store for a file, the driver or client will reassemble the chunks as needed. You can perform range queries on files stored through GridFS. You also can access information from arbitrary sections of files, which allows you to “skip” into the middle of a video or audio file.

GridFS is useful not only for storing files that exceed 16MB but also for storing any files for which you want access without having to load the entire file into memory. For more information on the indications of GridFS, see [When should I use GridFS?](#) (page 578).

Implement GridFS

To store and retrieve files using *GridFS*, use either of the following:

- A MongoDB driver. See the [drivers](#) (page 92) documentation for information on using GridFS with your driver.
- The `mongofiles` command-line tool in the `mongo` shell. See <http://docs.mongodb.org/manual/reference/program/mongofiles>.

GridFS Collections

GridFS stores files in two collections:

- `chunks` stores the binary chunks. For details, see [The chunks Collection](#) (page 125).
- `files` stores the file’s metadata. For details, see [The files Collection](#) (page 126).

⁴ The use of the term *chunks* in the context of GridFS is not related to the use of the term *chunks* in the context of sharding.

GridFS places the collections in a common bucket by prefixing each with the bucket name. By default, GridFS uses two collections with names prefixed by `fs` bucket:

- `fs.files`
- `fs.chunks`

You can choose a different bucket name than `fs`, and create multiple buckets in a single database.

Each document in the `chunks` collection represents a distinct chunk of a file as represented in the GridFS store. Each chunk is identified by its unique `ObjectID` stored in its `_id` field.

For descriptions of all fields in the `chunks` and `files` collections, see [GridFS Reference](#) (page 125).

GridFS Index

GridFS uses a *unique, compound* index on the `chunks` collection for the `files_id` and `n` fields. The `files_id` field contains the `_id` of the chunk’s “parent” document. The `n` field contains the sequence number of the chunk. GridFS numbers all chunks, starting with 0. For descriptions of the documents and fields in the `chunks` collection, see [GridFS Reference](#) (page 125).

The GridFS index allows efficient retrieval of chunks using the `files_id` and `n` values, as shown in the following example:

```
cursor = db.fs.chunks.find({files_id: myFileID}).sort({n:1});
```

See the relevant [driver](#) (page 92) documentation for the specific behavior of your GridFS application. If your driver does not create this index, issue the following operation using the mongo shell:

```
db.fs.chunks.ensureIndex( { files_id: 1, n: 1 }, { unique: true } );
```

Example Interface

The following is an example of the GridFS interface in Java. The example is for demonstration purposes only. For API specifics, see the relevant [driver](#) (page 92) documentation.

By default, the interface must support the default GridFS bucket, named `fs`, as in the following:

```
// returns default GridFS bucket (i.e. "fs" collection)
GridFS myFS = new GridFS(myDatabase);

// saves the file to "fs" GridFS bucket
myFS.createFile(new File("/tmp/largething.mpg"));
```

Optionally, interfaces may support other additional GridFS buckets as in the following example:

```
// returns GridFS bucket named "contracts"
GridFS myContracts = new GridFS(myDatabase, "contracts");

// retrieve GridFS object "smithco"
GridFSDBFile file = myContracts.findOne("smithco");

// saves the GridFS file to the file system
file.writeTo(new File("/tmp/smithco.pdf"));
```

3.3 Data Model Examples and Patterns

The following documents provide overviews of various data modeling patterns and common schema design considerations:

[Model Relationships Between Documents \(page 104\)](#) Examples for modeling relationships between documents.

[Model One-to-One Relationships with Embedded Documents \(page 104\)](#) Presents a data model that uses *embedded documents* (page 98) to describe one-to-one relationships between connected data.

[Model One-to-Many Relationships with Embedded Documents \(page 105\)](#) Presents a data model that uses *embedded documents* (page 98) to describe one-to-many relationships between connected data.

[Model One-to-Many Relationships with Document References \(page 106\)](#) Presents a data model that uses *references* (page 99) to describe one-to-many relationships between documents.

[Model Tree Structures \(page 108\)](#) Examples for modeling tree structures.

[Model Tree Structures with Parent References \(page 110\)](#) Presents a data model that organizes documents in a tree-like structure by storing *references* (page 99) to “parent” nodes in “child” nodes.

[Model Tree Structures with Child References \(page 111\)](#) Presents a data model that organizes documents in a tree-like structure by storing *references* (page 99) to “child” nodes in “parent” nodes.

See [Model Tree Structures](#) (page 108) for additional examples of data models for tree structures.

[Model Specific Application Contexts \(page 117\)](#) Examples for models for specific application contexts.

[Model Data for Atomic Operations \(page 117\)](#) Illustrates how embedding fields related to an atomic update within the same document ensures that the fields are in sync

[Model Data to Support Keyword Search \(page 118\)](#) Describes one method for supporting keyword search by storing keywords in an array in the same document as the text field. Combined with a multi-key index, this pattern can support application’s keyword search operations

3.3.1 Model Relationships Between Documents

[Model One-to-One Relationships with Embedded Documents \(page 104\)](#) Presents a data model that uses *embedded documents* (page 98) to describe one-to-one relationships between connected data.

[Model One-to-Many Relationships with Embedded Documents \(page 105\)](#) Presents a data model that uses *embedded documents* (page 98) to describe one-to-many relationships between connected data.

[Model One-to-Many Relationships with Document References \(page 106\)](#) Presents a data model that uses *references* (page 99) to describe one-to-many relationships between documents.

Model One-to-One Relationships with Embedded Documents

Overview

Data in MongoDB has a *flexible schema*. Collections do not enforce *document* structure. Decisions that affect how you model data can affect application performance and database capacity. See [Data Modeling Concepts](#) (page 97) for a full high level overview of data modeling in MongoDB.

This document describes a data model that uses *embedded* (page 98) documents to describe relationships between connected data.

Pattern

Consider the following example that maps patron and address relationships. The example illustrates the advantage of embedding over referencing if you need to view one data entity in context of the other. In this one-to-one relationship between patron and address data, the address belongs to the patron.

In the normalized data model, the address document contains a reference to the patron document.

```
{
  _id: "joe",
  name: "Joe Bookreader"
}

{
  patron_id: "joe",
  street: "123 Fake Street",
  city: "Faketown",
  state: "MA",
  zip: 12345
}
```

If the address data is frequently retrieved with the name information, then with referencing, your application needs to issue multiple queries to resolve the reference. The better data model would be to embed the address data in the patron data, as in the following document:

```
{
  _id: "joe",
  name: "Joe Bookreader",
  address: {
    street: "123 Fake Street",
    city: "Faketown",
    state: "MA",
    zip: 12345
  }
}
```

With the embedded data model, your application can retrieve the complete patron information with one query.

Model One-to-Many Relationships with Embedded Documents

Overview

Data in MongoDB has a *flexible schema*. Collections do not enforce *document* structure. Decisions that affect how you model data can affect application performance and database capacity. See [Data Modeling Concepts](#) (page 97) for a full high level overview of data modeling in MongoDB.

This document describes a data model that uses *embedded* (page 98) documents to describe relationships between connected data.

Pattern

Consider the following example that maps patron and multiple address relationships. The example illustrates the advantage of embedding over referencing if you need to view many data entities in context of another. In this one-to-many relationship between patron and address data, the patron has multiple address entities.

In the normalized data model, the address documents contain a reference to the patron document.

```
{  
  _id: "joe",  
  name: "Joe Bookreader"  
}  
  
{  
  patron_id: "joe",  
  street: "123 Fake Street",  
  city: "Faketown",  
  state: "MA",  
  zip: 12345  
}  
  
{  
  patron_id: "joe",  
  street: "1 Some Other Street",  
  city: "Boston",  
  state: "MA",  
  zip: 12345  
}
```

If your application frequently retrieves the address data with the name information, then your application needs to issue multiple queries to resolve the references. A more optimal schema would be to embed the address data entities in the patron data, as in the following document:

```
{  
  _id: "joe",  
  name: "Joe Bookreader",  
  addresses: [  
    {  
      street: "123 Fake Street",  
      city: "Faketown",  
      state: "MA",  
      zip: 12345  
    },  
    {  
      street: "1 Some Other Street",  
      city: "Boston",  
      state: "MA",  
      zip: 12345  
    }  
  ]  
}
```

With the embedded data model, your application can retrieve the complete patron information with one query.

Model One-to-Many Relationships with Document References

Overview

Data in MongoDB has a *flexible schema*. Collections do not enforce *document* structure. Decisions that affect how you model data can affect application performance and database capacity. See [Data Modeling Concepts](#) (page 97) for a full high level overview of data modeling in MongoDB.

This document describes a data model that uses [references](#) (page 99) between documents to describe relationships between connected data.

Pattern

Consider the following example that maps publisher and book relationships. The example illustrates the advantage of referencing over embedding to avoid repetition of the publisher information.

Embedding the publisher document inside the book document would lead to **repetition** of the publisher data, as the following documents show:

```
{
  title: "MongoDB: The Definitive Guide",
  author: [ "Kristina Chodorow", "Mike Dirolf" ],
  published_date: ISODate("2010-09-24"),
  pages: 216,
  language: "English",
  publisher: {
    name: "O'Reilly Media",
    founded: 1980,
    location: "CA"
  }
}

{
  title: "50 Tips and Tricks for MongoDB Developer",
  author: "Kristina Chodorow",
  published_date: ISODate("2011-05-06"),
  pages: 68,
  language: "English",
  publisher: {
    name: "O'Reilly Media",
    founded: 1980,
    location: "CA"
  }
}
```

To avoid repetition of the publisher data, use *references* and keep the publisher information in a separate collection from the book collection.

When using references, the growth of the relationships determine where to store the reference. If the number of books per publisher is small with limited growth, storing the book reference inside the publisher document may sometimes be useful. Otherwise, if the number of books per publisher is unbounded, this data model would lead to mutable, growing arrays, as in the following example:

```
{
  name: "O'Reilly Media",
  founded: 1980,
  location: "CA",
  books: [12346789, 234567890, ...]
}

{
  _id: 123456789,
  title: "MongoDB: The Definitive Guide",
  author: [ "Kristina Chodorow", "Mike Dirolf" ],
  published_date: ISODate("2010-09-24"),
  pages: 216,
  language: "English"
}
```

```
_id: 234567890,  
title: "50 Tips and Tricks for MongoDB Developer",  
author: "Kristina Chodorow",  
published_date: ISODate("2011-05-06"),  
pages: 68,  
language: "English"  
}
```

To avoid mutable, growing arrays, store the publisher reference inside the book document:

```
{  
  _id: "oreilly",  
  name: "O'Reilly Media",  
  founded: 1980,  
  location: "CA"  
}  
  
{  
  _id: 123456789,  
  title: "MongoDB: The Definitive Guide",  
  author: [ "Kristina Chodorow", "Mike Dirolf" ],  
  published_date: ISODate("2010-09-24"),  
  pages: 216,  
  language: "English",  
  publisher_id: "oreilly"  
}  
  
{  
  _id: 234567890,  
  title: "50 Tips and Tricks for MongoDB Developer",  
  author: "Kristina Chodorow",  
  published_date: ISODate("2011-05-06"),  
  pages: 68,  
  language: "English",  
  publisher_id: "oreilly"  
}
```

3.3.2 Model Tree Structures

MongoDB allows various ways to use tree data structures to model large hierarchical or nested data relationships.

Model Tree Structures with Parent References (page 110) Presents a data model that organizes documents in a tree-like structure by storing *references* (page 99) to “parent” nodes in “child” nodes.

Model Tree Structures with Child References (page 111) Presents a data model that organizes documents in a tree-like structure by storing *references* (page 99) to “child” nodes in “parent” nodes.

Model Tree Structures with an Array of Ancestors (page 113) Presents a data model that organizes documents in a tree-like structure by storing *references* (page 99) to “parent” nodes and an array that stores all ancestors.

Model Tree Structures with Materialized Paths (page 114) Presents a data model that organizes documents in a tree-like structure by storing full relationship paths between documents. In addition to the tree node, each document stores the `_id` of the nodes ancestors or path as a string.

Model Tree Structures with Nested Sets (page 116) Presents a data model that organizes documents in a tree-like structure using the *Nested Sets* pattern. This optimizes discovering subtrees at the expense of tree mutability.

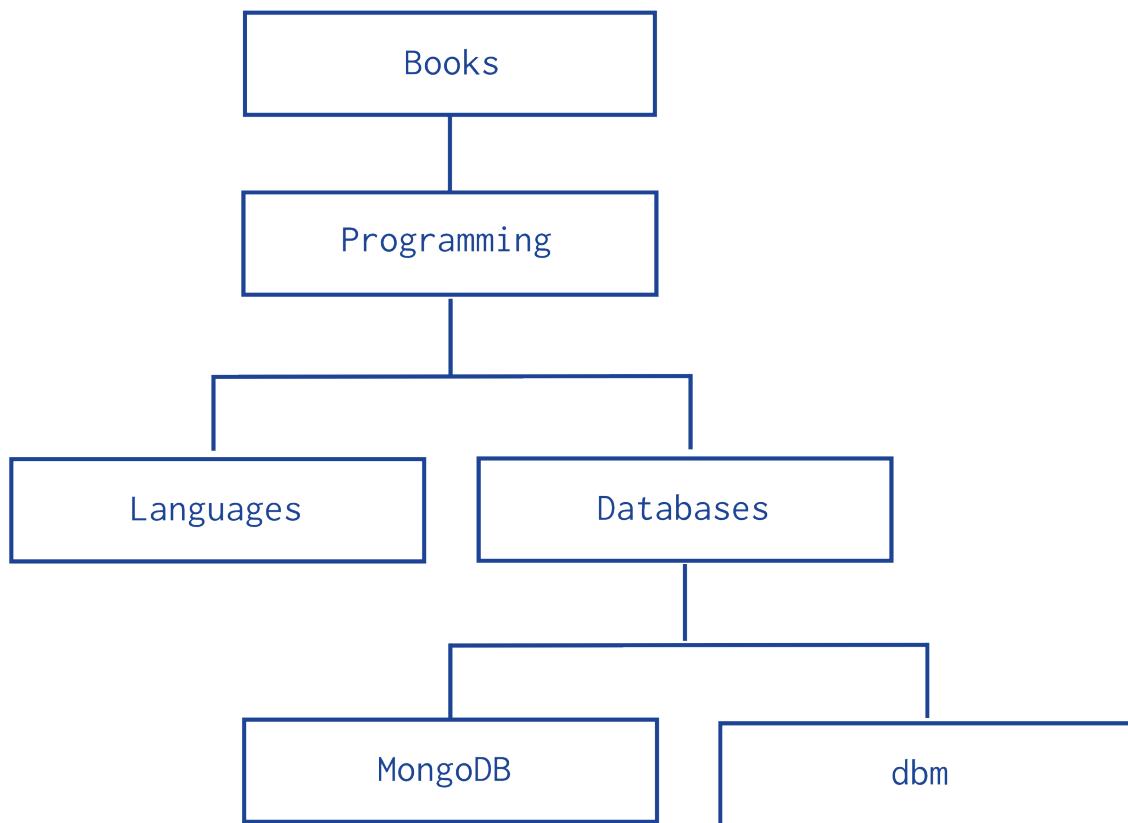


Figure 3.5: Tree data model for a sample hierarchy of categories.

Model Tree Structures with Parent References

Overview

Data in MongoDB has a *flexible schema*. Collections do not enforce *document* structure. Decisions that affect how you model data can affect application performance and database capacity. See [Data Modeling Concepts](#) (page 97) for a full high level overview of data modeling in MongoDB.

This document describes a data model that describes a tree-like structure in MongoDB documents by storing *references* (page 99) to “parent” nodes in children nodes.

Pattern

The *Parent References* pattern stores each tree node in a document; in addition to the tree node, the document stores the id of the node’s parent.

Consider the following hierarchy of categories:

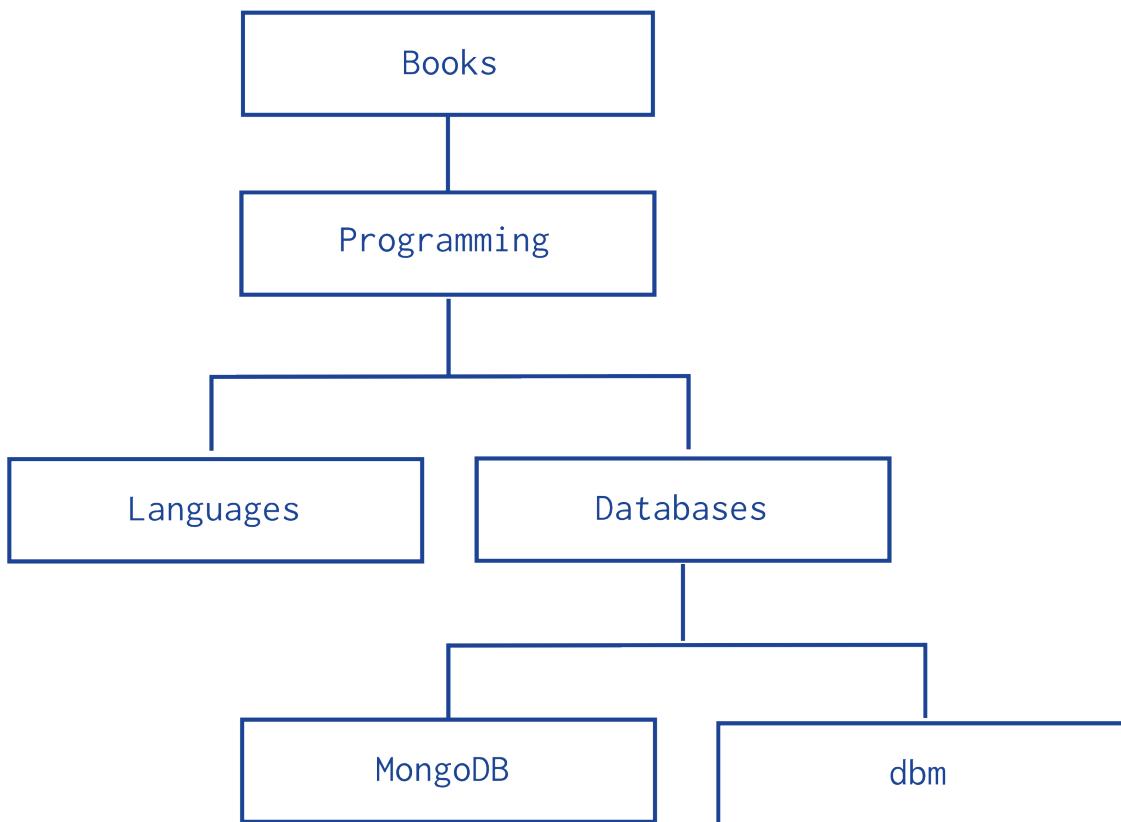


Figure 3.6: Tree data model for a sample hierarchy of categories.

The following example models the tree using *Parent References*, storing the reference to the parent category in the field `parent`:

```
db.categories.insert( { _id: "MongoDB", parent: "Databases" } )
db.categories.insert( { _id: "Postgres", parent: "Databases" } )
db.categories.insert( { _id: "Databases", parent: "Programming" } )
```

```
db.categories.insert( { _id: "Languages", parent: "Programming" } )
db.categories.insert( { _id: "Programming", parent: "Books" } )
db.categories.insert( { _id: "Books", parent: null } )
```

- The query to retrieve the parent of a node is fast and straightforward:

```
db.categories.findOne( { _id: "MongoDB" } ).parent
```

- You can create an index on the field `parent` to enable fast search by the parent node:

```
db.categories.ensureIndex( { parent: 1 } )
```

- You can query by the `parent` field to find its immediate children nodes:

```
db.categories.find( { parent: "Databases" } )
```

The *Parent Links* pattern provides a simple solution to tree storage but requires multiple queries to retrieve subtrees.

Model Tree Structures with Child References

Overview

Data in MongoDB has a *flexible schema*. *Collections* do not enforce *document* structure. Decisions that affect how you model data can affect application performance and database capacity. See [Data Modeling Concepts](#) (page 97) for a full high level overview of data modeling in MongoDB.

This document describes a data model that describes a tree-like structure in MongoDB documents by storing *references* (page 99) in the parent-nodes to children nodes.

Pattern

The *Child References* pattern stores each tree node in a document; in addition to the tree node, document stores in an array the id(s) of the node's children.

Consider the following hierarchy of categories:

The following example models the tree using *Child References*, storing the reference to the node's children in the field `children`:

```
db.categories.insert( { _id: "MongoDB", children: [] } )
db.categories.insert( { _id: "Postgres", children: [] } )
db.categories.insert( { _id: "Databases", children: [ "MongoDB", "Postgres" ] } )
db.categories.insert( { _id: "Languages", children: [] } )
db.categories.insert( { _id: "Programming", children: [ "Databases", "Languages" ] } )
db.categories.insert( { _id: "Books", children: [ "Programming" ] } )
```

- The query to retrieve the immediate children of a node is fast and straightforward:

```
db.categories.findOne( { _id: "Databases" } ).children
```

- You can create an index on the field `children` to enable fast search by the child nodes:

```
db.categories.ensureIndex( { children: 1 } )
```

- You can query for a node in the `children` field to find its parent node as well as its siblings:

```
db.categories.find( { children: "MongoDB" } )
```

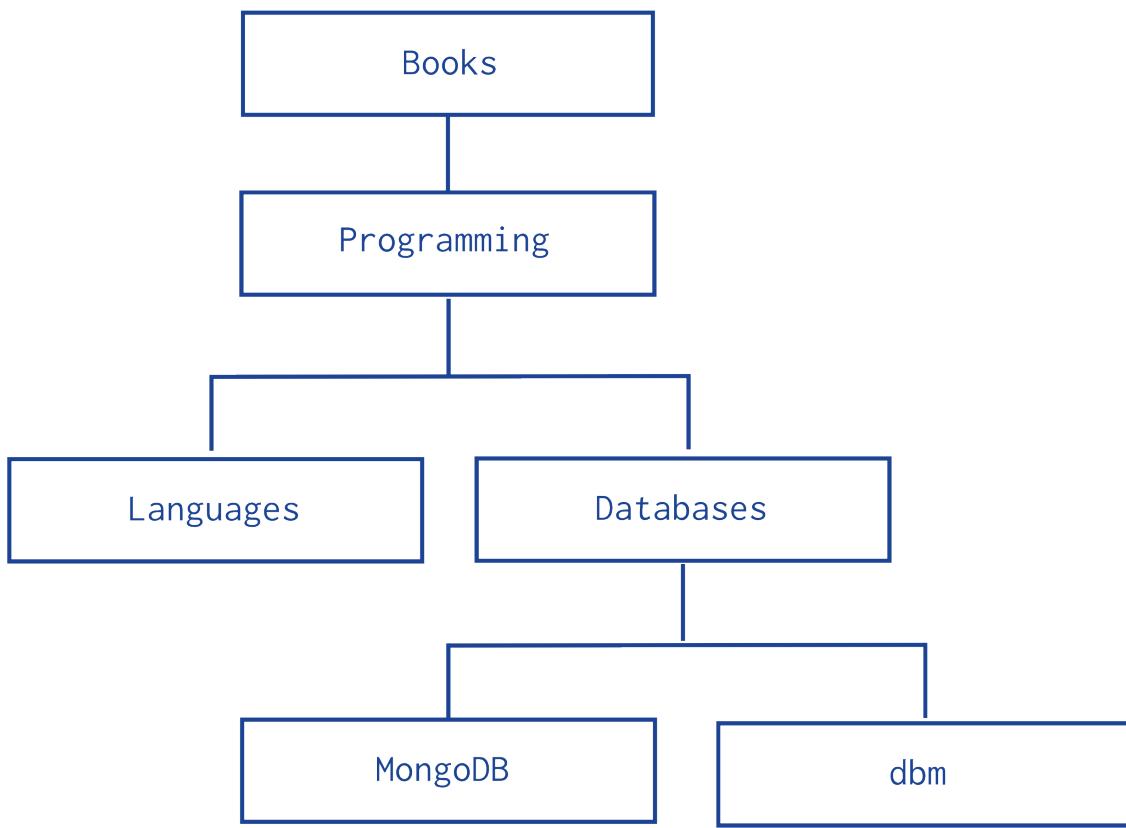


Figure 3.7: Tree data model for a sample hierarchy of categories.

The *Child References* pattern provides a suitable solution to tree storage as long as no operations on subtrees are necessary. This pattern may also provide a suitable solution for storing graphs where a node may have multiple parents.

Model Tree Structures with an Array of Ancestors

Overview

Data in MongoDB has a *flexible schema*. Collections do not enforce *document* structure. Decisions that affect how you model data can affect application performance and database capacity. See [Data Modeling Concepts](#) (page 97) for a full high level overview of data modeling in MongoDB.

This document describes a data model that describes a tree-like structure in MongoDB documents using [references](#) (page 99) to parent nodes and an array that stores all ancestors.

Pattern

The *Array of Ancestors* pattern stores each tree node in a document; in addition to the tree node, document stores in an array the id(s) of the node's ancestors or path.

Consider the following hierarchy of categories:

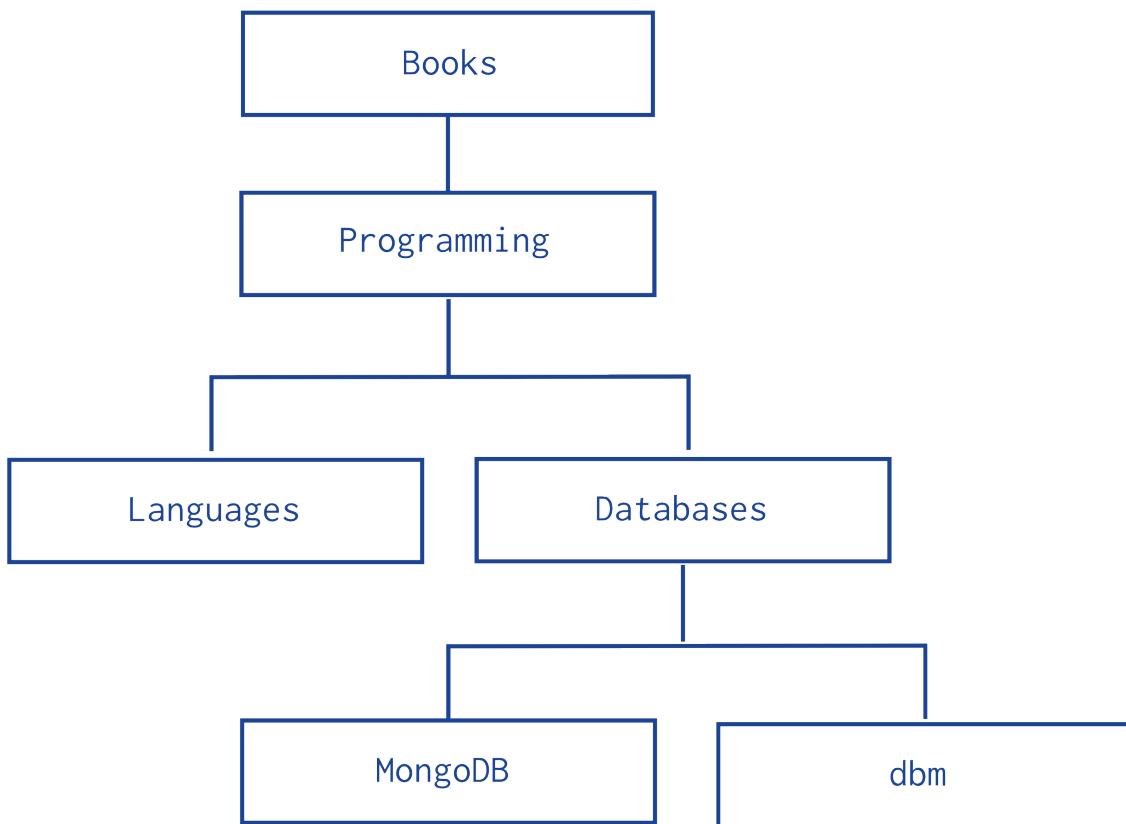


Figure 3.8: Tree data model for a sample hierarchy of categories.

The following example models the tree using *Array of Ancestors*. In addition to the `ancestors` field, these documents also store the reference to the immediate parent category in the `parent` field:

```
db.categories.insert( { _id: "MongoDB", ancestors: [ "Books", "Programming", "Databases" ], parent: null } )
db.categories.insert( { _id: "Postgres", ancestors: [ "Books", "Programming", "Databases" ], parent: null } )
db.categories.insert( { _id: "Databases", ancestors: [ "Books", "Programming" ], parent: "Programming" } )
db.categories.insert( { _id: "Languages", ancestors: [ "Books", "Programming" ], parent: "Programming" } )
db.categories.insert( { _id: "Programming", ancestors: [ "Books" ], parent: "Books" } )
db.categories.insert( { _id: "Books", ancestors: [ ], parent: null } )
```

- The query to retrieve the ancestors or path of a node is fast and straightforward:

```
db.categories.findOne( { _id: "MongoDB" } ).ancestors
```

- You can create an index on the field `ancestors` to enable fast search by the ancestors nodes:

```
db.categories.ensureIndex( { ancestors: 1 } )
```

- You can query by the field `ancestors` to find all its descendants:

```
db.categories.find( { ancestors: "Programming" } )
```

The *Array of Ancestors* pattern provides a fast and efficient solution to find the descendants and the ancestors of a node by creating an index on the elements of the `ancestors` field. This makes *Array of Ancestors* a good choice for working with subtrees.

The *Array of Ancestors* pattern is slightly slower than the *Materialized Paths* (page 114) pattern but is more straightforward to use.

Model Tree Structures with Materialized Paths

Overview

Data in MongoDB has a *flexible schema*. Collections do not enforce *document* structure. Decisions that affect how you model data can affect application performance and database capacity. See *Data Modeling Concepts* (page 97) for a full high level overview of data modeling in MongoDB.

This document describes a data model that describes a tree-like structure in MongoDB documents by storing full relationship paths between documents.

Pattern

The *Materialized Paths* pattern stores each tree node in a document; in addition to the tree node, document stores as a string the id(s) of the node's ancestors or path. Although the *Materialized Paths* pattern requires additional steps of working with strings and regular expressions, the pattern also provides more flexibility in working with the path, such as finding nodes by partial paths.

Consider the following hierarchy of categories:

The following example models the tree using *Materialized Paths*, storing the path in the field `path`; the path string uses the comma `,` as a delimiter:

```
db.categories.insert( { _id: "Books", path: null } )
db.categories.insert( { _id: "Programming", path: ",Books," } )
db.categories.insert( { _id: "Databases", path: ",Books,Programming," } )
db.categories.insert( { _id: "Languages", path: ",Books,Programming," } )
db.categories.insert( { _id: "MongoDB", path: ",Books,Programming,Databases," } )
db.categories.insert( { _id: "Postgres", path: ",Books,Programming,Databases," } )
```

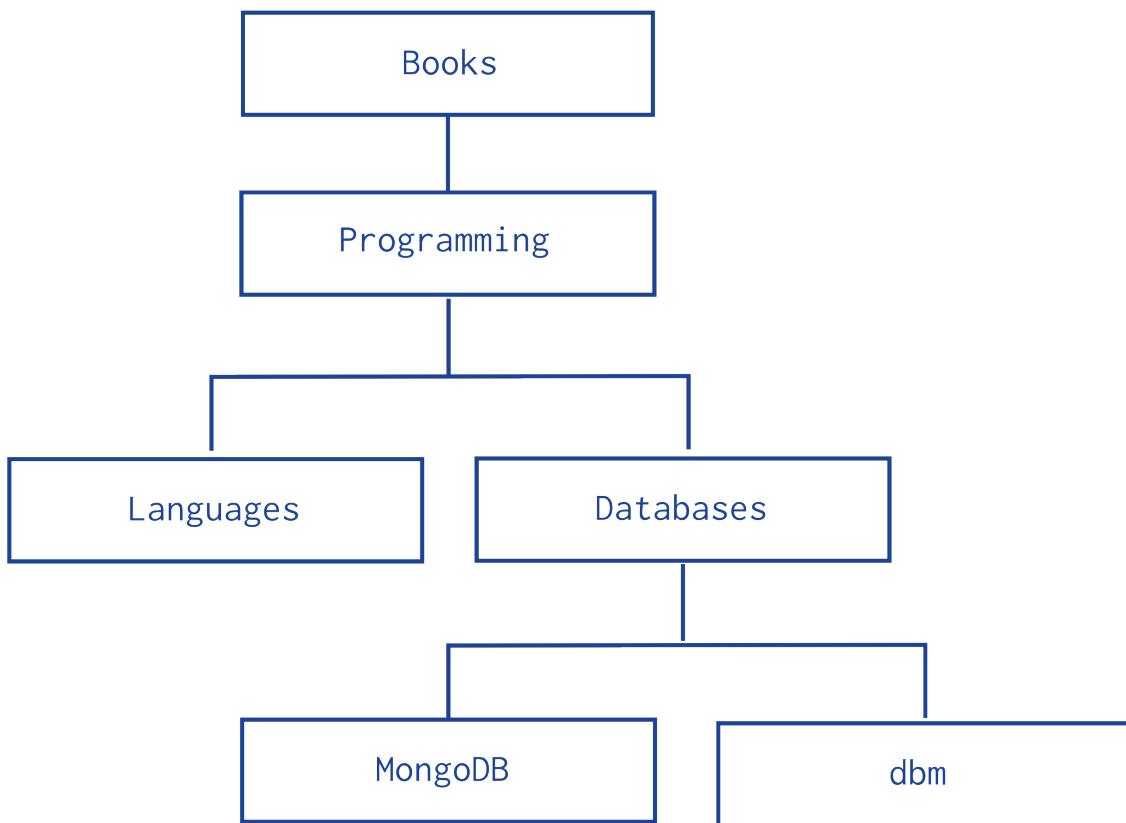


Figure 3.9: Tree data model for a sample hierarchy of categories.

- You can query to retrieve the whole tree, sorting by the field path:

```
db.categories.find().sort( { path: 1 } )
```

- You can use regular expressions on the path field to find the descendants of Programming:

```
db.categories.find( { path: /,Programming,/ } )
```

- You can also retrieve the descendants of Books where the Books is also at the topmost level of the hierarchy:

```
db.categories.find( { path: /^,Books,/ } )
```

- To create an index on the field path use the following invocation:

```
db.categories.ensureIndex( { path: 1 } )
```

This index may improve performance depending on the query:

- For queries of the Books sub-tree (e.g. `http://docs.mongodb.org/manual^,Books,/`) an index on the path field improves the query performance significantly.
- For queries of the Programming sub-tree (e.g. `http://docs.mongodb.org/manual,Programming,/`), or similar queries of sub-trees, where the node might be in the middle of the indexed string, the query must inspect the entire index.

For these queries an index *may* provide some performance improvement *if* the index is significantly smaller than the entire collection.

Model Tree Structures with Nested Sets

Overview

Data in MongoDB has a *flexible schema*. Collections do not enforce *document* structure. Decisions that affect how you model data can affect application performance and database capacity. See [Data Modeling Concepts](#) (page 97) for a full high level overview of data modeling in MongoDB.

This document describes a data model that describes a tree like structure that optimizes discovering subtrees at the expense of tree mutability.

Pattern

The *Nested Sets* pattern identifies each node in the tree as stops in a round-trip traversal of the tree. The application visits each node in the tree twice; first during the initial trip, and second during the return trip. The *Nested Sets* pattern stores each tree node in a document; in addition to the tree node, document stores the id of node's parent, the node's initial stop in the `left` field, and its return stop in the `right` field.

Consider the following hierarchy of categories:

The following example models the tree using *Nested Sets*:

```
db.categories.insert( { _id: "Books", parent: 0, left: 1, right: 12 } )
db.categories.insert( { _id: "Programming", parent: "Books", left: 2, right: 11 } )
db.categories.insert( { _id: "Languages", parent: "Programming", left: 3, right: 4 } )
db.categories.insert( { _id: "Databases", parent: "Programming", left: 5, right: 10 } )
db.categories.insert( { _id: "MongoDB", parent: "Databases", left: 6, right: 7 } )
db.categories.insert( { _id: "Postgres", parent: "Databases", left: 8, right: 9 } )
```

You can query to retrieve the descendants of a node:

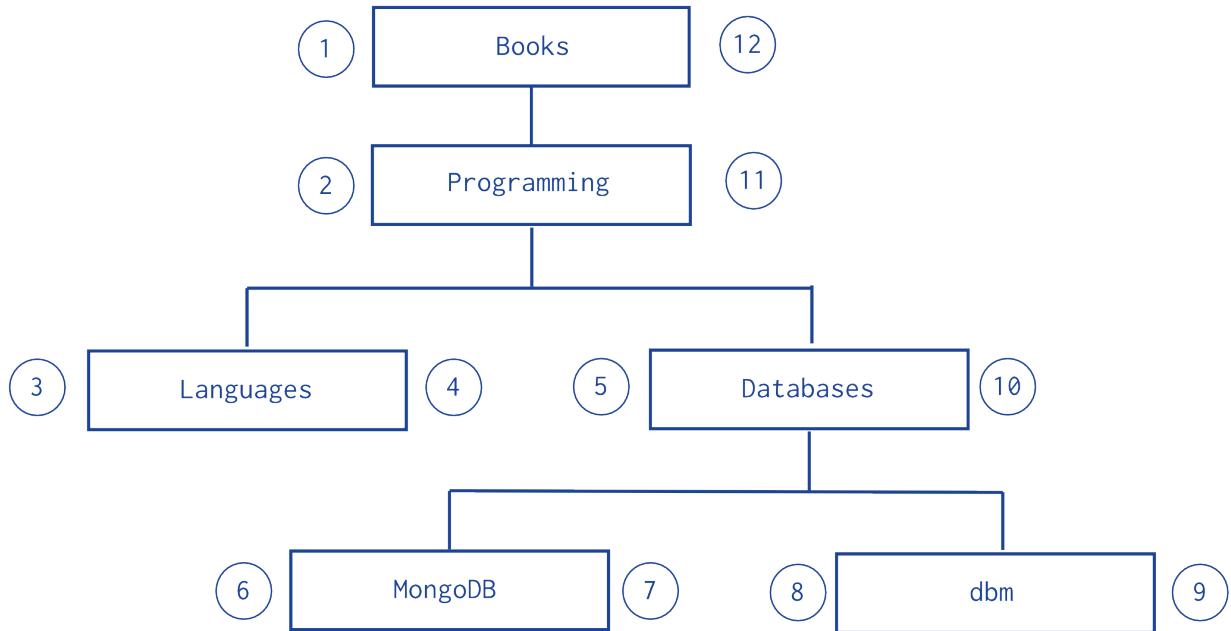


Figure 3.10: Example of a hierarchical data. The numbers identify the stops at nodes during a roundtrip traversal of a tree.

```
var databaseCategory = db.categories.findOne( { _id: "Databases" } );
db.categories.find( { left: { $gt: databaseCategory.left }, right: { $lt: databaseCategory.right } }
```

The *Nested Sets* pattern provides a fast and efficient solution for finding subtrees but is inefficient for modifying the tree structure. As such, this pattern is best for static trees that do not change.

3.3.3 Model Specific Application Contexts

[Model Data for Atomic Operations \(page 117\)](#) Illustrates how embedding fields related to an atomic update within the same document ensures that the fields are in sync

[Model Data to Support Keyword Search \(page 118\)](#) Describes one method for supporting keyword search by storing keywords in an array in the same document as the text field. Combined with a multi-key index, this pattern can support application's keyword search operations

Model Data for Atomic Operations

Pattern

Consider the following example that keeps a library book and its checkout information. The example illustrates how embedding fields related to an atomic update within the same document ensures that the fields are in sync.

Consider the following book document that stores the number of available copies for checkout and the current check-out information:

```
book = {
    _id: 123456789,
    title: "MongoDB: The Definitive Guide",
```

```
author: [ "Kristina Chodorow", "Mike Dirolf" ],
published_date: ISODate("2010-09-24"),
pages: 216,
language: "English",
publisher_id: "oreilly",
available: 3,
checkout: [ { by: "joe", date: ISODate("2012-10-15") } ]
}
```

You can use the `db.collection.findAndModify()` method to atomically determine if a book is available for checkout and update with the new checkout information. Embedding the `available` field and the `checkout` field within the same document ensures that the updates to these fields are in sync:

```
db.books.findAndModify ( {
  query: {
    _id: 123456789,
    available: { $gt: 0 }
  },
  update: {
    $inc: { available: -1 },
    $push: { checkout: { by: "abc", date: new Date() } }
  }
} )
```

Model Data to Support Keyword Search

Note: Keyword search is *not* the same as text search or full text search, and does not provide stemming or other text-processing features. See the [Limitations of Keyword Indexes](#) (page 119) section for more information.

In 2.4, MongoDB provides a text search feature. See [Text Indexes](#) (page 328) for more information.

If your application needs to perform queries on the content of a field that holds text you can perform exact matches on the text or use `$regex` to use regular expression pattern matches. However, for many operations on text, these methods do not satisfy application requirements.

This pattern describes one method for supporting keyword search using MongoDB to support application search functionality, that uses keywords stored in an array in the same document as the text field. Combined with a [multi-key index](#) (page 320), this pattern can support application's keyword search operations.

Pattern

To add structures to your document to support keyword-based queries, create an array field in your documents and add the keywords as strings in the array. You can then create a [multi-key index](#) (page 320) on the array and create queries that select values from the array.

Example

Given a collection of library volumes that you want to provide topic-based search. For each volume, you add the array `topics`, and you add as many keywords as needed for a given volume.

For the *Moby-Dick* volume you might have the following document:

```
{ title : "Moby-Dick" ,
  author : "Herman Melville" ,
  published : 1851 ,
  ISBN : 0451526996 ,
```

```

topics : [ "whaling" , "allegory" , "revenge" , "American" ,
  "novel" , "nautical" , "voyage" , "Cape Cod" ]
}

```

You then create a multi-key index on the `topics` array:

```
db.volumes.ensureIndex( { topics: 1 } )
```

The multi-key index creates separate index entries for each keyword in the `topics` array. For example the index contains one entry for `whaling` and another for `allegory`.

You then query based on the keywords. For example:

```
db.volumes.findOne( { topics : "voyage" } , { title: 1 } )
```

Note: An array with a large number of elements, such as one with several hundreds or thousands of keywords will incur greater indexing costs on insertion.

Limitations of Keyword Indexes

MongoDB can support keyword searches using specific data models and [multi-key indexes](#) (page 320); however, these keyword indexes are not sufficient or comparable to full-text products in the following respects:

- *Stemming*. Keyword queries in MongoDB can not parse keywords for root or related words.
- *Synonyms*. Keyword-based search features must provide support for synonym or related queries in the application layer.
- *Ranking*. The keyword look ups described in this document do not provide a way to weight results.
- *Asynchronous Indexing*. MongoDB builds indexes synchronously, which means that the indexes used for keyword indexes are always current and can operate in real-time. However, asynchronous bulk indexes may be more efficient for some kinds of content and workloads.

3.4 Data Model Reference

[Documents](#) (page 120) MongoDB stores all data in documents, which are JSON-style data structures composed of field-and-value pairs.

[Database References](#) (page 122) Discusses manual references and DBRefs, which MongoDB can use to represent relationships between documents.

[GridFS Reference](#) (page 125) Convention for storing large files in a MongoDB Database.

[ObjectId](#) (page 127) A 12-byte BSON type that MongoDB uses as the default value for its documents' `_id` field if the `_id` field is not specified.

[BSON Types](#) (page 129) Outlines the unique *BSON* types used by MongoDB. See [BSONspec.org](http://bsonspec.org)⁵ for the complete BSON specification.

⁵<http://bsonspec.org/>

3.4.1 Documents

MongoDB stores all data in documents, which are JSON-style data structures composed of field-and-value pairs:

```
{ "item": "pencil", "qty": 500, "type": "no.2" }
```

Most user-accessible data structures in MongoDB are documents, including:

- All database records.
- *Query selectors* (page 29), which define what records to select for read, update, and delete operations.
- *Update definitions* (page 40), which define what fields to modify during an update.
- *Index specifications* (page 314), which define what fields to index.
- Data output by MongoDB for reporting and configuration, such as the output of the `serverStatus` and the *replica set configuration document* (page 474).

Document Format

MongoDB stores documents on disk in the *BSON* serialization format. BSON is a binary representation of *JSON* documents, though contains more data types than does JSON. For the BSON spec, see bsonspec.org⁶. See also *BSON Types* (page 129).

The `mongo` JavaScript shell and the *MongoDB language drivers* (page 92) translate between BSON and the language-specific document representation.

Document Structure

MongoDB documents are composed of field-and-value pairs and have the following structure:

```
{
    field1: value1,
    field2: value2,
    field3: value3,
    ...
    fieldN: valueN
}
```

The value of a field can be any of the BSON *data types* (page 129), including other documents, arrays, and arrays of documents. The following document contains values of varying types:

```
var mydoc = {
    _id: ObjectId("5099803df3f4948bd2f98391"),
    name: { first: "Alan", last: "Turing" },
    birth: new Date('Jun 23, 1912'),
    death: new Date('Jun 07, 1954'),
    contribs: [ "Turing machine", "Turing test", "Turingery" ],
    views : NumberLong(1250000)
}
```

The above fields have the following data types:

- `_id` holds an *ObjectId*.
- `name` holds a *subdocument* that contains the fields `first` and `last`.

⁶<http://bsonspec.org/>

- birth and death hold values of the *Date* type.
- contribs holds an *array of strings*.
- views holds a value of the *NumberLong* type.

Field Names

Field names are strings. Field names cannot contain null characters, dots (.) or dollar signs (\$). See [Dollar Sign Operator Escaping](#) (page 579) for an alternate approach.

BSON documents may have more than one field with the same name. Most [MongoDB interfaces](#) (page 92), however, represent MongoDB with a structure (e.g. a hash table) that does not support duplicate field names. If you need to manipulate documents that have more than one field with the same name, see the [driver documentation](#) (page 92) for your driver.

Some documents created by internal MongoDB processes may have duplicate fields, but *no* MongoDB process will ever add duplicate fields to an existing user document.

Document Limitations

Documents have the following attributes:

- The maximum BSON document size is 16 megabytes.

The maximum document size helps ensure that a single document cannot use excessive amount of RAM or, during transmission, excessive amount of bandwidth. To store documents larger than the maximum size, MongoDB provides the GridFS API. See `mongofiles` and the documentation for your [driver](#) (page 92) for more information about GridFS.

- [Documents](#) (page 120) have the following restrictions on field names:
 - The field name `_id` is reserved for use as a primary key; its value must be unique in the collection, is immutable, and may be of any type other than an array.
 - The field names **cannot** start with the \$ character.
 - The field names **cannot** contain the . character.
- MongoDB does not make guarantees regarding the order of fields in a BSON document. Drivers and MongoDB will reorder the fields of a documents upon insertion and following updates.

Most programming languages represent BSON documents with some form of *mapping type*. Comparisons between mapping type objects typically, depend on order. As a result, the only way to ensure that two documents have the same set of field and value pairs is to compare each field and value individually.

The `_id` Field

The `_id` field has the following behavior and constraints:

- In documents, the `_id` field is always indexed for regular collections.
- The `_id` field may contain values of any [BSON data type](#) (page 129), other than an array.

Warning: To ensure functioning replication, do not store values that are of the BSON regular expression type in the `_id` field.

The following are common options for storing values for `_id`:

- Use an *ObjectId* (page 127).
- Use a natural unique identifier, if available. This saves space and avoids an additional index.
- Generate an auto-incrementing number. See *Create an Auto-Incrementing Sequence Field* (page 75).
- Generate a UUID in your application code. For a more efficient storage of the UUID values in the collection and in the `_id` index, store the UUID as a value of the BSON `BinData` type.

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.
- Use your driver's BSON UUID facility to generate UUIDs. Be aware that driver implementations may implement UUID serialization and deserialization logic differently, which may not be fully compatible with other drivers. See your driver documentation⁷ for information concerning UUID interoperability.

Note: Most MongoDB driver clients will include the `_id` field and generate an `ObjectId` before sending the insert operation to MongoDB; however, if the client sends a document without an `_id` field, the `mongod` will add the `_id` field and generate the `ObjectId`.

Dot Notation

MongoDB uses the *dot notation* to access the elements of an array and to access the fields of a subdocument.

To access an element of an array by the zero-based index position, concatenate the array name with the dot (.) and zero-based index position, and enclose in quotes:

```
'<array>. <index>'
```

To access a field of a subdocument with *dot-notation*, concatenate the subdocument name with the dot (.) and the field name, and enclose in quotes:

```
'<subdocument>. <field>'
```

See also:

- *Subdocuments* (page 59) for dot notation examples with subdocuments.
- *Arrays* (page 59) for dot notation examples with arrays.

3.4.2 Database References

MongoDB does not support joins. In MongoDB some data is *denormalized*, or stored with related data in *documents* to remove the need for joins. However, in some cases it makes sense to store related information in separate documents, typically in different collections or databases.

MongoDB applications use one of two methods for relating documents:

1. *Manual references* (page 123) where you save the `_id` field of one document in another document as a reference. Then your application can run a second query to return the embedded data. These references are simple and sufficient for most use cases.
2. *DBRefs* (page 124) are references from one document to another using the value of the first document's `_id` field collection, and optional database name. To resolve DBRefs, your application must perform additional

⁷<http://api.mongodb.org/>

queries to return the referenced documents. Many [drivers](#) (page 92) have helper methods that form the query for the DBRef automatically. The drivers ⁸ do not *automatically* resolve DBRefs into documents.

Use a DBRef when you need to embed documents from multiple collections in documents from one collection. DBRefs also provide a common format and type to represent these relationships among documents. The DBRef format provides common semantics for representing links between documents if your database must interact with multiple frameworks and tools.

Unless you have a compelling reason for using a DBRef, use manual references.

Manual References

Background

Manual references refers to the practice of including one *document's* `_id` field in another document. The application can then issue a second query to resolve the referenced fields as needed.

Process

Consider the following operation to insert two documents, using the `_id` field of the first document as a reference in the second document:

```
original_id = ObjectId()

db.places.insert({
    "_id": original_id,
    "name": "Broadway Center",
    "url": "bc.example.net"
})

db.people.insert({
    "name": "Erin",
    "places_id": original_id,
    "url": "bc.example.net/Erin"
})
```

Then, when a query returns the document from the `people` collection you can, if needed, make a second query for the document referenced by the `places_id` field in the `places` collection.

Use

For nearly every case where you want to store a relationship between two documents, use [manual references](#) (page 123). The references are simple to create and your application can resolve references as needed.

The only limitation of manual linking is that these references do not convey the database and collection name. If you have documents in a single collection that relate to documents in more than one collection, you may need to consider using [DBRefs](#) (page 124).

⁸ Some community supported drivers may have alternate behavior and may resolve a DBRef into a document automatically.

DBRefs

Background

DBRefs are a convention for representing a *document*, rather than a specific reference “type.” They include the name of the collection, and in some cases the database, in addition to the value from the `_id` field.

Format

DBRefs have the following fields:

`$ref`

The `$ref` field holds the name of the collection where the referenced document resides.

`$id`

The `$id` field contains the value of the `_id` field in the referenced document.

`$db`

Optional.

Contains the name of the database where the referenced document resides.

Only some drivers support `$db` references.

Example

DBRef document would resemble the following:

```
{ "$ref" : <value>, "$id" : <value>, "$db" : <value> }
```

Consider a document from a collection that stored a DBRef in a `creator` field:

```
{
  "_id" : ObjectId("5126bbf64aed4daf9e2ab771"),
  // .. application fields
  "creator" : {
    "$ref" : "creators",
    "$id" : ObjectId("5126bc054aed4daf9e2ab772"),
    "$db" : "users"
  }
}
```

The DBRef in this example, points to a document in the `creators` collection of the `users` database that has `ObjectId("5126bc054aed4daf9e2ab772")` in its `_id` field.

Note: The order of fields in the DBRef matters, and you must use the above sequence when using a DBRef.

Support

C++ The C++ driver contains no support for DBRefs. You can transverse references manually.

C# The C# driver provides access to DBRef objects with the [MongoDBRef Class⁹](#) and supplies the [FetchDBRef Method¹⁰](#) for accessing these objects.

⁹<http://api.mongodb.org/csharp/current/html/46c356d3-ed06-a6f8-42fa-e0909ab64ce2.htm>

¹⁰<http://api.mongodb.org/csharp/current/html/1b0b8f48-ba98-1367-0a7d-6e01c8df436f.htm>

Java The `DBRef`¹¹ class provides supports for DBRefs from Java.

JavaScript The mongo shell's JavaScript interface provides a `DBRef`.

Perl The Perl driver contains no support for DBRefs. You can transverse references manually or use the `MongoDBx::AutoDeref`¹² CPAN module.

PHP The PHP driver does support DBRefs, including the optional `$db` reference, through `The MongoDBRef class`¹³.

Python The Python driver provides the `DBRef` class¹⁴, and the `dereference` method¹⁵ for interacting with DBRefs.

Ruby The Ruby Driver supports DBRefs using the `DBRef` class¹⁶ and the `deference` method¹⁷.

Use

In most cases you should use the *manual reference* (page 123) method for connecting two or more related documents. However, if you need to reference documents from multiple collections, consider a `DBRef`.

3.4.3 GridFS Reference

GridFS stores files in two collections:

- `chunks` stores the binary chunks. For details, see *The chunks Collection* (page 125).
- `files` stores the file's metadata. For details, see *The files Collection* (page 126).

GridFS places the collections in a common bucket by prefixing each with the bucket name. By default, GridFS uses two collections with names prefixed by `fs` bucket:

- `fs.files`
- `fs.chunks`

You can choose a different bucket name than `fs`, and create multiple buckets in a single database.

See also:

GridFS (page 102) for more information about GridFS.

The chunks Collection

Each document in the `chunks` collection represents a distinct chunk of a file as represented in the *GridFS* store. The following is a prototype document from the `chunks` collection.:

```
{
  "_id" : <ObjectId>,
  "files_id" : <ObjectID>,
  "n" : <num>,
  "data" : <binary>
}
```

A document from the `chunks` collection contains the following fields:

¹¹<http://api.mongodb.org/java/current/com/mongodb/DBRef.html>

¹²<http://search.cpan.org/dist/MongoDBx-AutoDeref/>

¹³<http://www.php.net/manual/en/class.mongodb.bref.php>

¹⁴<http://api.mongodb.org/python/current/api/bson/dbref.html>

¹⁵<http://api.mongodb.org/python/current/api/pymongo/database.html#pymongo.database.Database.dereference>

¹⁶<http://api.mongodb.org/ruby/current/BSON/DBRef.html>

¹⁷<http://api.mongodb.org/ruby/current/Mongo/DB.html#dereference>

chunks._id

The unique *ObjectID* of the chunk.

chunks.files_id

The *_id* of the “parent” document, as specified in the *files* collection.

chunks.n

The sequence number of the chunk. GridFS numbers all chunks, starting with 0.

chunks.data

The chunk’s payload as a *BSON* binary type.

The *chunks* collection uses a *compound index* on *files_id* and *n*, as described in [GridFS Index](#) (page 103).

The *files* Collection

Each document in the *files* collection represents a file in the *GridFS* store. Consider the following prototype of a document in the *files* collection:

```
{  
  "_id" : <ObjectID>,  
  "length" : <num>,  
  "chunkSize" : <num>  
  "uploadDate" : <timestampl>  
  "md5" : <hash>  
  
  "filename" : <string>,  
  "contentType" : <string>,  
  "aliases" : <string array>,  
  "metadata" : <dataObject>,  
}
```

Documents in the *files* collection contain some or all of the following fields. Applications may create additional arbitrary fields:

files._id

The unique ID for this document. The *_id* is of the data type you chose for the original document. The default type for MongoDB documents is *BSON ObjectID*.

files.length

The size of the document in bytes.

files.chunkSize

The size of each chunk. GridFS divides the document into chunks of the size specified here. The default size is 256 kilobytes.

files.uploadDate

The date the document was first stored by GridFS. This value has the *Date* type.

files.md5

An MD5 hash returned from the *filemd5* API. This value has the *String* type.

files.filename

Optional. A human-readable name for the document.

files.contentType

Optional. A valid MIME type for the document.

files.aliases

Optional. An array of alias strings.

files.metadata

Optional. Any additional information you want to store.

3.4.4 ObjectId

Overview

ObjectId is a 12-byte *BSON* type, constructed using:

- a 4-byte value representing the seconds since the Unix epoch,
- a 3-byte machine identifier,
- a 2-byte process id, and
- a 3-byte counter, starting with a random value.

In MongoDB, documents stored in a collection require a unique `_id` field that acts as a *primary key*. Because ObjectIds are small, most likely unique, and fast to generate, MongoDB uses ObjectIds as the default value for the `_id` field if the `_id` field is not specified. MongoDB clients should add an `_id` field with a unique ObjectId. However, if a client does not add an `_id` field, mongod will add an `_id` field that holds an ObjectId.

Using ObjectIds for the `_id` field provides the following additional benefits:

- in the mongo shell, you can access the creation time of the ObjectId, using the `getTimestamp()` method.
- sorting on an `_id` field that stores ObjectId values is roughly equivalent to sorting by creation time.

Important: The relationship between the order of ObjectId values and generation time is not strict within a single second. If multiple systems, or multiple processes or threads on a single system generate values, within a single second; ObjectId values do not represent a strict insertion order. Clock skew between clients can also result in non-strict ordering even for values, because client drivers generate ObjectId values, *not* the mongod process.

Also consider the [Documents](#) (page 120) section for related information on MongoDB's document orientation.

ObjectId()

The mongo shell provides the `ObjectId()` wrapper class to generate a new ObjectId, and to provide the following helper attribute and methods:

- `str`
The hexadecimal string value of the `ObjectId()` object.
- `getTimestamp()`
Returns the timestamp portion of the `ObjectId()` object as a Date.
- `toString()`
Returns the string representation of the `ObjectId()` object. The returned string literal has the format “`ObjectId(...)`”.
- Changed in version 2.2: In previous versions `toString()` returns the value of the ObjectId as a hexadecimal string.
- `valueOf()`

Returns the value of the `ObjectId()` object as a hexadecimal string. The returned string is the `str` attribute.

Changed in version 2.2: In previous versions `valueOf()` returns the `ObjectId()` object.

Examples

Consider the following uses `ObjectId()` class in the mongo shell:

Generate a new ObjectId

To generate a new ObjectId, use the `ObjectId()` constructor with no argument:

```
x = ObjectId()
```

In this example, the value of `x` would be:

```
ObjectId("507f1f77bcf86cd799439011")
```

To generate a new ObjectId using the `ObjectId()` constructor with a unique hexadecimal string:

```
y = ObjectId("507f191e810c19729de860ea")
```

In this example, the value of `y` would be:

```
ObjectId("507f191e810c19729de860ea")
```

- To return the timestamp of an `ObjectId()` object, use the `getTimestamp()` method as follows:

Convert an ObjectId into a Timestamp

To return the timestamp of an `ObjectId()` object, use the `getTimestamp()` method as follows:

```
ObjectId("507f191e810c19729de860ea").getTimestamp()
```

This operation will return the following Date object:

```
ISODate("2012-10-17T20:46:22Z")
```

Convert ObjectIds into Strings

Access the `str` attribute of an `ObjectId()` object, as follows:

```
ObjectId("507f191e810c19729de860ea").str
```

This operation will return the following hexadecimal string:

```
507f191e810c19729de860ea
```

To return the value of an `ObjectId()` object as a hexadecimal string, use the `valueOf()` method as follows:

```
ObjectId("507f191e810c19729de860ea").valueOf()
```

This operation returns the following output:

507f191e810c19729de860ea

To return the string representation of an `ObjectId()` object, use the `toString()` method as follows:

```
ObjectId("507f191e810c19729de860ea").toString()
```

This operation will return the following output:

```
ObjectId("507f191e810c19729de860ea")
```

3.4.5 BSON Types

- [ObjectId](#) (page 130)
- [String](#) (page 130)
- [Timestamps](#) (page 130)
- [Date](#) (page 131)

BSON is a binary serialization format used to store documents and make remote procedure calls in MongoDB. The *BSON* specification is located at bsonspec.org¹⁸.

BSON supports the following data types as values in documents. Each data type has a corresponding number that can be used with the `$type` operator to query documents by *BSON* type.

Type	Number
Double	1
String	2
Object	3
Array	4
Binary data	5
Object id	7
Boolean	8
Date	9
Null	10
Regular Expression	11
JavaScript	13
Symbol	14
JavaScript (with scope)	15
32-bit integer	16
Timestamp	17
64-bit integer	18
Min key	255
Max key	127

When comparing values of different *BSON* types, MongoDB uses the following comparison order, from lowest to highest:

1. MinKey (internal type)
2. Null
3. Numbers (ints, longs, doubles)
4. Symbol, String
5. Object

¹⁸<http://bsonspec.org/>

6. Array
7. BinData
8. ObjectId
9. Boolean
10. Date, Timestamp
11. Regular Expression
12. MaxKey (internal type)

Note: MongoDB treats some types as equivalent for comparison purposes. For instance, numeric types undergo conversion before comparison.

To determine a field's type, see [Check Types in the mongo Shell](#) (page 205).

If you convert BSON to JSON, see [Data Type Fidelity](#) (page 148) for more information.

The next sections describe special considerations for particular BSON types.

ObjectId

ObjectIds are: small, likely unique, fast to generate, and ordered. These values consists of 12-bytes, where the first four bytes are a timestamp that reflect the ObjectId's creation. Refer to the [ObjectId](#) (page 127) documentation for more information.

String

BSON strings are UTF-8. In general, drivers for each programming language convert from the language's string format to UTF-8 when serializing and deserializing BSON. This makes it possible to store most international characters in BSON strings with ease.¹⁹ In addition, MongoDB \$regex queries support UTF-8 in the regex string.

Timestamps

BSON has a special timestamp type for *internal* MongoDB use and is **not** associated with the regular [Date](#) (page 131) type. Timestamp values are a 64 bit value where:

- the first 32 bits are a `time_t` value (seconds since the Unix epoch)
- the second 32 bits are an incrementing `ordinal` for operations within a given second.

Within a single mongod instance, timestamp values are always unique.

In replication, the `oplog` has a `ts` field. The values in this field reflect the operation time, which uses a BSON timestamp value.

Note: The BSON Timestamp type is for *internal* MongoDB use. For most cases, in application development, you will want to use the BSON date type. See [Date](#) (page 131) for more information.

If you create a BSON Timestamp using the empty constructor (e.g. `new Timestamp()`), MongoDB will only generate a timestamp *if* you use the constructor in the first field of the document.²⁰ Otherwise, MongoDB will generate an empty timestamp value (i.e. `Timestamp(0, 0)`.)

¹⁹ Given strings using UTF-8 character sets, using `sort()` on strings will be reasonably correct. However, because internally `sort()` uses the C++ `strcmp` api, the sort order may handle some characters incorrectly.

²⁰ If the first field in the document is `_id`, then you can generate a timestamp in the `second` field of a document.

Changed in version 2.1: mongo shell displays the Timestamp value with the wrapper:

```
Timestamp(<time_t>, <ordinal>)
```

Prior to version 2.1, the mongo shell display the Timestamp value as a document:

```
{ t : <time_t>, i : <ordinal> }
```

Date

BSON Date is a 64-bit integer that represents the number of milliseconds since the Unix epoch (Jan 1, 1970). The official BSON specification²¹ refers to the BSON Date type as the *UTC datetime*.

Changed in version 2.0: BSON Date type is signed.²² Negative values represent dates before 1970.

Example

Construct a Date using the new Date() constructor in the mongo shell:

```
var mydate1 = new Date()
```

Example

Construct a Date using the ISODate() constructor in the mongo shell:

```
var mydate2 = ISODate()
```

Example

Return the Date value as string:

```
mydate1.toString()
```

Example

Return the month portion of the Date value; months are zero-indexed, so that January is month 0:

```
mydate1.getMonth()
```

²¹<http://bsonspec.org/#/specification>

²² Prior to version 2.0, Date values were incorrectly interpreted as *unsigned* integers, which affected sorts, range queries, and indexes on Date fields. Because indexes are not recreated when upgrading, please re-index if you created an index on Date values with an earlier version, and dates before 1970 are relevant to your application.

Administration

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.

***Administration Concepts* (page 133)** Core conceptual documentation of operational practices for managing MongoDB deployments and systems.

***Backup Strategies for MongoDB Systems* (page 134)** Describes approaches and considerations for backing up a MongoDB database.

***Data Center Awareness* (page 157)** 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.

***Monitoring for MongoDB* (page 136)** An overview of monitoring tools, diagnostic strategies, and approaches to monitoring replica sets and sharded clusters.

***Administration Tutorials* (page 167)** Tutorials that describe common administrative procedures and practices for operations for MongoDB instances and deployments.

***Configuration, Maintenance, and Analysis* (page 167)** Describes routine management operations, including configuration and performance analysis.

***Backup and Recovery* (page 184)** Outlines procedures for data backup and restoration with mongod instances and deployments.

***Administration Reference* (page 219)** Reference and documentation of internal mechanics of administrative features, systems and functions and operations.

See also:

The MongoDB Manual contains administrative documentation and tutorials though out several sections. See *Replica Set Tutorials* (page 415) and *Sharded Cluster Tutorials* (page 513) for additional tutorials and information.

4.1 Administration Concepts

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

***Operational Strategies* (page 134)** Higher level documentation of key concepts for the operation and maintenance of MongoDB deployments, including backup, maintenance, and configuration.

***Backup Strategies for MongoDB Systems* (page 134)** Describes approaches and considerations for backing up a MongoDB database.

Monitoring for MongoDB (page 136) An overview of monitoring tools, diagnostic strategies, and approaches to monitoring replica sets and sharded clusters.

Run-time Database Configuration (page 143) Outlines common MongoDB configurations and examples of best-practice configurations for common use cases.

Data Management (page 156) Core documentation that addresses issues in data management, organization, maintenance, and lifestyle management.

Data Center Awareness (page 157) 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.

Expire Data from Collections by Setting TTL (page 160) 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.

Capped Collections (page 158) Capped collections provide a special type of size-constrained collections that preserve insertion order and can support high volume inserts.

Optimization Strategies for MongoDB (page 162) Techniques for optimizing application performance with MongoDB.

4.1.1 Operational Strategies

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

Backup Strategies for MongoDB Systems (page 134) Describes approaches and considerations for backing up a MongoDB database.

Monitoring for MongoDB (page 136) An overview of monitoring tools, diagnostic strategies, and approaches to monitoring replica sets and sharded clusters.

Run-time Database Configuration (page 143) Outlines common MongoDB configurations and examples of best-practice configurations for common use cases.

Import and Export MongoDB Data (page 147) Provides an overview of mongoimport and mongoexport, the tools MongoDB includes for importing and exporting data.

Production Notes (page 151) A collection of notes that describe best practices and considerations for the operations of MongoDB instances and deployments.

Backup Strategies for MongoDB Systems

Backups are an important part of any operational disaster recovery plan. A good backup plan must be able to capture data in a consistent and usable state, and operators must be able to automate both the backup and the recovery operations. Also test all components of the backup system to ensure that you can recover backed up data as needed. If you cannot effectively restore your database from the backup, then your backups are useless.

Note: The [MongoDB Management Service](#)¹ supports backup and restoration for MongoDB deployments. See the [MMS Backup Documentation](#)² for more information.

¹https://mms.10gen.com/?pk_campaign=MongoDB-Org&pk_kwd=Backup-Docs

²<https://mms.mongodb.com/help/backup/>

Backup Considerations

As you develop a backup strategy for your MongoDB deployment consider the following factors:

- Geography. Ensure that you move some backups away from your primary database infrastructure.
- System errors. Ensure that your backups can survive situations where hardware failures or disk errors impact the integrity or availability of your backups.
- Production constraints. Backup operations themselves sometimes require substantial system resources. It is important to consider the time of the backup schedule relative to peak usage and maintenance windows.
- System capabilities. Some of the block-level snapshot tools require special support on the operating-system or infrastructure level.
- Database configuration. *Replication* and *sharding* can affect the process and impact of the backup implementation. See [Sharded Cluster Backup Considerations](#) (page 135) and [Replica Set Backup Considerations](#) (page 136).
- Actual requirements. You may be able to save time, effort, and space by including only crucial data in the most frequent backups and backing up less crucial data less frequently.

Warning: In order to use filesystem snapshots for backups, your `mongod` instance must have `journal` enabled, which is the default for 64 bit versions of MongoDB since 2.0. If the journal sits on a different filesystem than your data files then you must also disable writes while the snapshot completes.

Backup Approaches

There are two main methodologies for backing up MongoDB instances. Creating binary “dumps” of the database using `mongodump` or creating filesystem level snapshots. Both methodologies have advantages and disadvantages:

- binary database dumps are comparatively small, because they don’t include index content or pre-allocated free space, and [record padding](#) (page 55). However, it’s impossible to capture a copy of a running system that reflects a single moment in time using a binary dump.
- filesystem snapshots, sometimes called block level backups, produce larger backup sizes, but complete quickly and can reflect a single moment in time on a running system. However, snapshot systems require filesystem and operating system support and tools.

The best option depends on the requirements of your deployment and disaster recovery needs. Typically, filesystem snapshots are because of their accuracy and simplicity; however, `mongodump` is a viable option used often to generate backups of MongoDB systems.

In some cases, taking backups is difficult or impossible because of large data volumes, distributed architectures, and data transmission speeds. In these situations, increase the number of members in your replica set or sets.

Backup and Recovery Procedures

For tutorials on the backup approaches, see [Backup and Recovery](#) (page 184).

Backup Strategies for MongoDB Deployments

Sharded Cluster Backup Considerations

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.

Sharded clusters complicate backup operations, as distributed systems. True point-in-time backups are only possible when stopping all write activity from the application. To create a precise moment-in-time snapshot of a cluster, stop all application write activity to the database, capture a backup, and allow only write operations to the database after the backup is complete.

However, you can capture a backup of a cluster that **approximates** a point-in-time backup by capturing a backup from a secondary member of the replica sets that provide the shards in the cluster at roughly the same moment. If you decide to use an approximate-point-in-time backup method, ensure that your application can operate using a copy of the data that does not reflect a single moment in time.

For backup procedures for sharded clusters, see [Backup and Restore Sharded Clusters](#) (page 193).

Replica Set Backup Considerations In most cases, backing up data stored in a *replica set* is similar to backing up data stored in a single instance. Options include:

- Create a file system snapshot of a single *secondary*, as described in [Backup and Restore with MongoDB Tools](#) (page 185). You may choose to maintain a dedicated *hidden member* for backup purposes.
- As an alternative you can create a backup with the `mongodump` program and the `--oplog` option. To restore this backup use the `mongorestore` program and the `--oplogReplay` option.

If you have a *sharded cluster* where each *shard* is itself a replica set, you can use one of these methods to create a backup of the entire cluster without disrupting the operation of the node. In these situations you should still turn off the balancer when you create backups.

For any cluster, using a non-primary node to create backups is particularly advantageous in that the backup operation does not affect the performance of the primary. Replication itself provides some measure of redundancy. Nevertheless, keeping point-in time backups of your cluster to provide for disaster recovery and as an additional layer of protection is crucial.

Monitoring for MongoDB

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 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 Management Service \(MMS\)](#)³ is a hosted monitoring service which collects and aggregates data to provide insight into the performance and operation of MongoDB deployments. See the [MMS documentation](#)⁴ 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, [MMS Monitoring Service](#)⁵ collects data from running MongoDB deployments and provides visualization and alerts based on that data. MMS is a free service provided by MongoDB.

³https://mms.mongodb.com/?pk_campaign=mongodb-org&pk_kwd=monitoring

⁴<http://mms.mongodb.com/help/>

⁵https://mms.mongodb.com/?pk_campaign=mongodb-org&pk_kwd=monitoring

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.

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.

REST Interface MongoDB provides a simple REST interface that can be useful for configuring monitoring and alert scripts, and for other administrative tasks.

To enable, configure `mongod` to use *REST*, either by starting `mongod` with the `--rest` option, or by setting the `rest` setting to `true` in a configuration file.

For more information on using the REST Interface see, the [Simple REST Interface⁶](#) documentation.

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.

⁶<http://docs.mongodb.org/ecosystem/tools/http-interfaces>

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 [MMS](#)⁷. 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` 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.

repSetGetStatus The `repSetGetStatus` command (`rs.status()` from the shell) returns an overview of your replica set's status. The `repSetGetStatus` 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.

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 ⁸	<code>mongodb-ganglia</code> ⁹	Python script to report operations per second, memory usage, btree statistics, master/slave status and current connections.
Ganglia	<code>gmond_python_modules</code> ¹⁰	Parses output from the <code>serverStatus</code> and <code>repSetGetStatus</code> commands.
Motop ¹¹	<i>None</i>	Realtime monitoring tool for MongoDB servers. Shows current operations ordered by durations every second.
mtop ¹²	<i>None</i>	A top like tool.
Munin ¹³	<code>mongo-munin</code> ¹⁴	Retrieves server statistics.
Munin	<code>mongomon</code> ¹⁵	Retrieves collection statistics (sizes, index sizes, and each (configured) collection count for one DB).
Munin	<code>munin-plugins Ubuntu PPA</code> ¹⁶	Some additional munin plugins not in the main distribution.
Nagios ¹⁷	<code>nagios-plugin-mongodb</code> ¹⁸	A simple Nagios check script, written in Python.
Zabbix ¹⁹	<code>mikoomi-mongodb</code> ²⁰	Monitors availability, resource utilization, health, performance and other important metrics.

⁷<http://mms.mongodb.com>

⁸<http://sourceforge.net/apps/trac/ganglia/wiki>

Also consider [dex²¹](#), an index and query analyzing tool for MongoDB that compares MongoDB log files and indexes to make indexing recommendations.

As part of [MongoDB Enterprise²²](#), you can run [MMS On-Prem²³](#), which offers the features of MMS in a package that runs within your infrastructure.

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

Name	Notes
MongoDB Management Service²⁴	MMS is a cloud-based suite of services for managing MongoDB deployments. MMS provides monitoring and backup functionality.
Scout²⁵	Several plugins, including MongoDB Monitoring²⁶ , MongoDB Slow Queries²⁷ , and MongoDB Replica Set Monitoring²⁸ .
Server Density²⁹	Dashboard for MongoDB³⁰ , MongoDB specific alerts, replication failover timeline and iPhone, iPad and Android mobile apps.

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.
- `verbose`. Increases the amount of information written to the log or output.

You can also specify this as `v` (as in `-v`). For higher levels of verbosity, set multiple `v`, as in `vvvv = True`. You can also change the verbosity of a running `mongod` or `mongos` instance with the `setParameter` command.

- `logpath`. 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:

⁹<https://github.com/quiver/mongodb-ganglia>

¹⁰https://github.com/ganglia/gmond_python_modules

¹¹<https://github.com/tart/motop>

¹²<https://github.com/beaufour/mtop>

¹³<http://munin-monitoring.org/>

¹⁴<https://github.com/erh/mongo-munin>

¹⁵<https://github.com/pcdummy/mongomon>

¹⁶<https://launchpad.net/ chris-lea/+archive/munin-plugins>

¹⁷<http://www.nagios.org/>

¹⁸<https://github.com/mzupan/nagios-plugin-mongodb>

¹⁹<http://www.zabbix.com/>

²⁰<https://code.google.com/p/mikoomi/wiki/03>

²¹<https://github.com/mongolab/dex>

²²<http://www.mongodb.com/products/mongodb-enterprise>

²³<http://mms.mongodb.com>

²⁴https://mms.mongodb.com/?pk_campaign=mongodb-org&pk_kwd=monitoring

²⁵<http://scoutapp.com>

²⁶https://scoutapp.com/plugin_urls/391-mongodb-monitoring

²⁷https://scoutapp.com/plugin_urls/291-mongodb-slow-queries

²⁸https://scoutapp.com/plugin_urls/2251-mongodb-replica-set-monitoring

²⁹<http://www.serverdensity.com>

³⁰<http://www.serverdensity.com/mongodb-monitoring/>

```
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 177).

Diagnosing Performance Issues

Degraded performance in MongoDB is typically a function of the relationship between the quantity of data stored in the database, the amount of system RAM, the number of connections to the database, and the amount of time the database spends in a locked state.

In some cases performance issues may be transient and related to traffic load, data access patterns, or the availability of hardware on the host system for virtualized environments. Some users also experience performance limitations as a result of inadequate or inappropriate indexing strategies, or as a consequence of poor schema design patterns. In other situations, performance issues may indicate that the database may be operating at capacity and that it is time to add additional capacity to the database.

The following are some causes of degraded performance in MongoDB.

Locks MongoDB uses a locking system to ensure consistency. However, if certain operations are long-running, or a queue forms, performance will slow as requests and operations wait for the lock. Lock-related slowdowns can be intermittent. To see if the lock has been affecting your performance, look to the data in the `globalLock` section of the `serverStatus` output. 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. If `globalLock.ratio` is also high, MongoDB has likely been processing a large number of long running queries. Long queries are often the result of a number of factors: ineffective use of indexes, non-optimal schema design, poor query structure, system architecture issues, or insufficient RAM resulting in [page faults](#) (page 140) and disk reads.

Memory Usage MongoDB uses memory mapped files to store data. Given a data set of sufficient size, the MongoDB process will allocate all available memory on the system for its use. While this is part of the design, and affords MongoDB superior performance, the memory mapped files make it difficult to determine if the amount of RAM is sufficient for the data set.

The *memory usage* metrics of the `serverStatus` output can provide insight into MongoDB's memory use. Check the resident memory use (i.e. `mem.resident`): 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 should also check the amount of mapped memory (i.e. `mem.mapped`). If this value is greater than the amount of system memory, some operations will require disk access *page faults* to read data from virtual memory and negatively affect performance.

Page Faults A page fault occurs when MongoDB requires data not located in physical memory, and must read from virtual memory. To check for page faults, see the `extra_info.page_faults` value in the `serverStatus` output. This data is only available on Linux systems.

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. In many situations, MongoDB’s read locks will “yield” after a page fault to allow other processes to read and avoid blocking while waiting for the next page to read into memory. This approach improves concurrency, and also improves overall throughput in high volume systems.

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

Number of Connections In some cases, the number of connections between the application layer (i.e. clients) and the database can overwhelm the ability of the server to handle requests. This can produce performance irregularities. 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 collections available for new clients.

Note: Unless constrained by system-wide limits MongoDB has a hard connection limit of 20,000 connections. You can modify system limits using the `ulimit` command, or by editing your system’s `/etc/sysctl` file.

If requests are high because 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.

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 `slowms` 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 `slowms` at runtime as an argument to the `db.setProfilingLevel()` operation.

See

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

By default, mongod records all “slow” queries to its log, as defined by slowms. Unlike log data, the data in system.profile does not persist between mongod restarts.

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 slowms threshold.

See also:

[Optimization Strategies for MongoDB](#) (page 162) addresses strategies that may improve the performance of your database queries and operations.

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 oplogSize in the MongoDB configuration file. If you do not specify this on the command line before running with the --repSet 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 441)

For causes of replication lag, see [Replication Lag](#) (page 458).

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 <http://docs.mongodb.org/manual/reference/command/replSetGetStatus> document 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 [Sharding Concepts](#) (page 492) 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.

[MMS Monitoring](#)³¹ monitors config servers and can create notifications if a config server becomes inaccessible.

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 insure 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.

Run-time Database Configuration

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 control script or installed from a package for your operating system, you likely already have a configuration file located at `/etc/c/mongodb.conf`. Confirm this by checking the contents of the `/etc/init.d/mongod` or `/etc/rc.d/mongod` script to insure that the *control scripts* start the `mongod` with the appropriate configuration file (see below.)

³¹<http://mms.mongodb.com>

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

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

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

Configure the Database

Consider the following basic configuration:

```
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.
- `bind_ip` 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”).
- `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. In normal operation this is the preferable operation to avoid log noise. In diagnostic or testing situations, set this value to `false`. 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.
- `logpath` 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.
- `journal` 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:

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

Consider the following explanation for these configuration decisions:

- “bind_ip” 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.

- “nounixsocket” to true disables the UNIX Socket, which is otherwise enabled by default. This limits access on the local system. This is desirable when running MongoDB on systems with shared access, but in most situations has minimal impact.
- “auth” is true enables the authentication 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:

[Security Concepts](#) (page 235)

Replication and Sharding Configuration

Replication Configuration *Replica set* configuration is straightforward, and only requires that the `rep1Set` have a value that is consistent among all members of the set. Consider the following:

```
rep1Set = set0
```

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

See also:

[Replica set reconfiguration](#) (page 477).

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

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

New in version 1.8: for replica sets, and 1.9.1 for sharded replica sets.

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 Reconfiguration](#) (page 477) section for information regarding the process for changing replica set during operation.

Additionally, consider the [Replica Set Security](#) (page 236) section for information on configuring authentication with replica sets.

Finally, see the [Replication](#) (page 373) document for more information on replication in MongoDB and replica set configuration in general.

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 144) mongod instances, and then add the following configuration option:

```
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 144), adding the shardsvr setting:

```
shardsvr = true
```

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

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

You can specify multiple configdb instances by specifying hostnames and ports in the form of a comma separated list. In general, avoid modifying the chunkSize from the default value of 64,³² and should ensure this setting is consistent among all mongos instances.

See also:

The [Sharding](#) (page 487) 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 deployments³³ 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 144) for each instance, but consider the following configuration values:

```
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 its *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 *control scripts* and/or adjust your existing MongoDB configuration and control script as needed to control these processes.

³² 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.

³³ 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.

Diagnostic Configurations

The following configuration options control various `mongod` behaviors for diagnostic purposes. The following settings have default values that tuned for general production purposes:

```
slowms = 50
profile = 3
verbose = true
diaglog = 3
objcheck = true
cpu = true
```

Use the [base configuration](#) (page 144) and add these options if you are experiencing some unknown issue or performance problem as needed:

- `slowms` configures the threshold for the *database profiler* to consider a query “slow.” The default value is 100 milliseconds. Set a lower value if the database profiler does not return useful results. See [Optimization Strategies for MongoDB](#) (page 162) for more information on optimizing operations in MongoDB.
- `profile` sets the *database profiler* level. The profiler is not active by default because of the possible impact on the profiler itself on performance. Unless this setting has a value, queries are not profiled.
- `verbose` enables a verbose logging mode that modifies `mongod` output and increases logging to include a greater number of events. Only use this option if you are experiencing an issue that is not reflected in the normal logging level. If you require additional verbosity, consider the following options:

```
v = true
vv = true
vvv = true
vvvv = true
vvvvv = true
```

Each additional level `v` adds additional verbosity to the logging. The `verbose` option is equal to `v = true`.

- `diaglog` enables *diagnostic logging*. Level 3 logs all read and write options.
- `objcheck` forces `mongod` to validate all requests from clients upon receipt. Use this option to ensure that invalid requests are not causing errors, particularly when running a database with untrusted clients. This option may affect database performance.
- `cpu` forces `mongod` to report the percentage of the last interval spent in *write lock*. The interval is typically 4 seconds, and each output line in the log includes both the actual interval since the last report and the percentage of time spent in write lock.

Import and Export MongoDB Data

This document provides an overview of the import and export programs included in the MongoDB distribution. These tools are useful when you want to backup or export a portion of your data without capturing the state of the entire database, or for simple data ingestion cases. 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 itself. For disaster recovery protection and routine database backup operation, use full [database instance backups](#) (page 134).

Warning: Because these tools primarily operate by interacting with a running `mongod` instance, they can impact the performance of your running database.

Not only do these processes 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.

`mongoimport` and `mongoexport` do not reliably preserve all rich *BSON* data types, because *BSON* is a superset of *JSON*. Thus, `mongoimport` and `mongoexport` cannot represent *BSON* data accurately in *JSON*. As a result data exported or imported with these tools may lose some measure of fidelity. See [MongoDB Extended JSON](#) (page 223) for more information about MongoDB Extended JSON.

See also:

See the [Backup Strategies for MongoDB Systems](#) (page 134) document for more information on backing up MongoDB instances. Additionally, consider the following references for commands addressed in this document:

- <http://docs.mongodb.org/manual/reference/program/mongoexport>
- <http://docs.mongodb.org/manual/reference/program/mongorestore>
- <http://docs.mongodb.org/manual/reference/program/mongodump>

If you want to transform and process data once you've imported it in MongoDB consider the documents in the [Aggregation](#) (page 273) section, including:

- [Map-Reduce](#) (page 280) and
- [Aggregation Concepts](#) (page 277).

Data Type Fidelity

JSON does not have the following data types that exist in *BSON* documents: `data_binary`, `data_date`, `data_timestamp`, `data_regex`, `data_oid` and `data_ref`. As a result using any tool that decodes *BSON* documents into *JSON* will suffer some loss of fidelity.

If maintaining type fidelity is important, consider writing a data import and export system that does not force *BSON* documents into *JSON* form as part of the process. The following list of types contain examples for how MongoDB will represent how *BSON* documents render in *JSON*.

- `data_binary`

```
{ "$binary" : "<bindata>", "$type" : "<t>" }
```

`<bindata>` is the base64 representation of a binary string. `<t>` is the hexadecimal representation of a single byte indicating the data type.

- `data_date`

```
Date( <date> )
```

`<date>` is the JSON representation of a 64-bit signed integer for milliseconds since epoch.

- `data_timestamp`

```
Timestamp( <t>, <i> )
```

`<t>` is the JSON representation of a 32-bit unsigned integer for milliseconds since epoch. `<i>` is a 32-bit unsigned integer for the increment.

- `data_regex`

/<jRegex>/<jOptions>

<jRegex> is a string that may contain valid JSON characters and unescaped double quote (i.e. ") characters, but may not contain unescaped forward slash (i.e. <http://docs.mongodb.org/manual>) characters. <jOptions> is a string that may contain only the characters g, i, m, and s.

- data_oid

`ObjectId("<id>")`

<id> is a 24 character hexadecimal string. These representations require that data_oid values have an associated field named “_id.”

- data_ref

`DBRef("<name>", "<id>")`

<name> is a string of valid JSON characters. <id> is a 24 character hexadecimal string.

See also:

[MongoDB Extended JSON](#) (page 223)

Data Import and Export and Backups Operations

For resilient and non-disruptive backups, use a file system or block-level disk snapshot function, such as the methods described in the [Backup Strategies for MongoDB Systems](#) (page 134) document. The tools and operations discussed provide functionality that’s useful in the context of providing some kinds of backups.

By contrast, use import and export tools to backup a small subset of your data or to move data to or from a 3rd party system. These backups may capture a small crucial set of data or a frequently modified section of data, for extra insurance, or for ease of access. No matter how you decide to import or export your data, consider the following guidelines:

- Label files so that you can identify what point in time the export or backup reflects.
- Labeling should describe the contents of the backup, and reflect the subset of the data corpus, captured in the backup or export.
- Do not create or apply exports if the backup process itself will have an adverse effect on a production system.
- Make sure that they 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.

Human Intelligible Import/Export Formats

This section describes a process to import/export your database, or a portion thereof, to a file in a *JSON* or *CSV* format.

See also:

The <http://docs.mongodb.org/manual/reference/program/mongoimport> and <http://docs.mongodb.org/manual/reference/program/mongoexport> documents contain complete documentation of these tools. If you have questions about the function and parameters of these tools not covered here, please refer to these documents.

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.

These tools may also be useful for importing data into a MongoDB database from third party applications.

Collection Export with mongoexport With the `mongoexport` utility you can create a backup file. In the most simple invocation, the command takes the following form:

```
mongoexport --collection collection --out collection.json
```

This will export all documents in the collection named `collection` into the file `collection.json`. Without the output specification (i.e. “`--out collection.json`”), `mongoexport` writes output to standard output (i.e. “`stdout`.”) You can further narrow the results by supplying a query filter using the “`--query`” and limit results to a single database using the “`--db`” option. For instance:

```
mongoexport --db sales --collection contacts --query '{"field": 1}'
```

This command returns all documents in the `sales` database’s `contacts` collection, with a field named `field` with a value of 1. Enclose the query in single quotes (e.g. ‘) to ensure that it does not interact with your shell environment. The resulting documents will return on standard output.

By default, `mongoexport` returns one *JSON document* per MongoDB document. Specify the “`--jsonArray`” argument to return the export as a single *JSON array*. Use the “`--csv`” file to return the result in CSV (comma separated values) format.

If your `mongod` instance is not running, you can use the “`--dbpath`” option to specify the location to your MongoDB instance’s database files. See the following example:

```
mongoexport --db sales --collection contacts --dbpath /srv/MongoDB/
```

This reads the data files directly. This locks the data directory to prevent conflicting writes. The `mongod` process must *not* be running or attached to these data files when you run `mongoexport` in this configuration.

The “`--host`” and “`--port`” options allow you to specify a non-local host to connect to capture the export. Consider the following example:

```
mongoexport --host mongodb1.example.net --port 37017 --username user --password pass --collection contacts
```

On any `mongoexport` command you may, as above specify username and password credentials as above.

Collection Import with mongoimport To restore a backup taken with `mongoexport`. Most of the arguments to `mongoexport` also exist for `mongoimport`. Consider the following command:

```
mongoimport --collection collection --file collection.json
```

This imports the contents of the file `collection.json` into the collection named `collection`. If you do not specify a file with the “`--file`” option, `mongoimport` accepts input over standard input (e.g. “`stdin`.”)

If you specify the “`--upsert`” option, all of `mongoimport` operations will attempt to update existing documents in the database and insert other documents. This option will cause some performance impact depending on your configuration.

You can specify the database option `--db` to import these documents to a particular database. If your MongoDB instance is not running, use the “`--dbpath`” option to specify the location of your MongoDB instance’s database files. Consider using the “`--journal`” option to ensure that `mongoimport` records its operations in the journal. The `mongod` process must *not* be running or attached to these data files when you run `mongoimport` in this configuration.

Use the “`--ignoreBlanks`” option to ignore blank fields. For CSV and TSV imports, this option provides the desired functionality in most cases: it avoids inserting blank fields in MongoDB documents.

Production Notes

Production Notes

- Packages (page 151)
 - MongoDB (page 151)
 - Operating Systems (page 152)
- Concurrency (page 152)
- Write Concern (page 152)
- Journaling (page 152)
- Connection Pool (page 152)
- Hardware Requirements and Limitations (page 152)
- MongoDB on NUMA Hardware (page 153)
- Disk and Storage Systems (page 153)
 - Swap (page 153)
 - RAID (page 154)
 - Remote Filesystems (page 154)
 - Separate Components onto Different Storage Devices (page 154)
- MongoDB on Linux (page 154)
 - Kernel and File Systems (page 154)
 - Recommended Configuration (page 155)
- Networking (page 155)
- MongoDB on Virtual Environments (page 155)
 - EC2 (page 155)
 - VMWare (page 156)
 - OpenVZ (page 156)
- Performance Monitoring (page 156)
 - iostat (page 156)
 - bwm-ng (page 156)
- Backups (page 156)

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

Note: MongoDB Management Service (MMS)³⁴ is a hosted monitoring service which collects and aggregates diagnostic data to provide insight into the performance and operation of MongoDB deployments. See the [MMS Website³⁵](#) and the [MMS documentation³⁶](#) for more information.

Packages

MongoDB Be sure you have the latest stable release. All releases are available on the [Downloads³⁷](#) page. This is a good place to verify what is current, even if you then choose to install via a package manager.

Always use 64-bit builds for production. The 32-bit build MongoDB offers for test and development environments is not suitable for production deployments as it can store no more than 2GB of data. See the [32-bit limitations](#) (page 574) for more information.

32-bit builds exist to support use on development machines.

³⁴<http://mms.mongodb.com>

³⁵<http://mms.mongodb.com/>

³⁶<http://mms.mongodb.com/help/>

³⁷<http://www.mongodb.org/downloads>

Operating Systems MongoDB distributions are currently available for Mac OS X, Linux, Windows Server 2008 R2 64bit, Windows 7 (32 bit and 64 bit), Windows Vista, and Solaris platforms.

Note: MongoDB uses the [GNU C Library](#)³⁸ (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.

Concurrency

In earlier versions of MongoDB, all write operations contended for a single readers-writer lock on the MongoDB instance. As of version 2.2, each database has a readers-writer lock that allows concurrent reads access to a database, but gives exclusive access to a single write operation per database. See the [Concurrency](#) (page 586) page for more information.

Write Concern

Write concern describes the guarantee that MongoDB provides when reporting on the success of a write operation. The strength of the write concerns determine the level of guarantee. When inserts, updates and deletes have a *weak* write concern, write operations return quickly. In some failure cases, write operations issued with weak write concerns may not persist. With *stronger* write concerns, clients wait after sending a write operation for MongoDB to confirm the write operations.

MongoDB provides different levels of write concern to better address the specific needs of applications. Clients may adjust write concern to ensure that the most important operations persist successfully to an entire MongoDB deployment. For other less critical operations, clients can adjust the write concern to ensure faster performance rather than ensure persistence to the entire deployment.

See [Write Concern](#) (page 44) for more information about choosing an appropriate write concern level for your deployment.

Journaling

MongoDB uses *write ahead logging* to an on-disk *journal* to guarantee that MongoDB is able to quickly recover the [write operations](#) (page 40) following a crash or other serious failure.

In order to ensure that mongod will be able to recover and remain in a consistent state following a crash, you should leave journaling enabled. See [Journaling](#) (page 229) for more information.

Connection Pool

To avoid overloading the connection resources of a single mongod or mongos instance, ensure that clients maintain reasonable connection pool sizes.

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

Hardware Requirements and Limitations

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.

³⁸<http://www.gnu.org/software/libc/>

The hardware for the most effective MongoDB deployments have the following properties:

- As with all software, more RAM and a faster CPU clock speed are important for performance.
- In general, databases are not CPU bound. As such, increasing the number of cores can help, but does not provide significant marginal return.
- 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 mongod.
- Commodity (SATA) spinning drives are often a good option, as the increase to random I/O for more expensive 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.
- Remote file storage can create performance problems in MongoDB. See [Remote Filesystems](#) (page 154) for more information about storage and MongoDB.

MongoDB on NUMA Hardware

Important: The discussion of NUMA in this section only applies to Linux, and therefore does not affect deployments where mongod instances run other UNIX-like systems or on Windows.

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 or high system process usage.

When running MongoDB on NUMA hardware, you should disable NUMA for MongoDB and instead set an interleave memory policy.

Note: MongoDB version 2.0 and greater checks these settings on start up when deployed on a Linux-based system, and prints a warning if the system is NUMA-based.

To disable NUMA for MongoDB and set an interleave memory policy, use the numactl command and start mongod in the following manner:

```
numactl --interleave=all /usr/bin/local/mongod
```

Then, disable *zone reclaim* in the `/proc` settings using the following command:

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

To fully disable NUMA, you must perform both operations. For more information, see the Documentation for `/proc/sys/vm/*`³⁹.

See the [The MySQL “swap insanity” problem and the effects of NUMA](#)⁴⁰ post, which describes the effects of NUMA on databases. This blog post addresses the impact of NUMA for MySQL, but the issues for MongoDB are similar. The post introduces NUMA and its goals, and illustrates how these goals are not compatible with production databases.

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.

³⁹<http://www.kernel.org/doc/Documentation/sysctl/vm.txt>

⁴⁰<http://jcole.us/blog/archives/2010/09/28/mysql-swap-insanity-and-the-numa-architecture/>

The method `mongod` uses to map memory files to memory ensures that the operating system will never store MongoDB 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.

Remote Filesystems The Network File System protocol (NFS) is not recommended for use with MongoDB as some versions perform poorly.

Performance problems arise 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. If you must 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.

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

MongoDB on Linux

Important: The following discussion only applies to Linux, and therefore does not affect deployments where `mongod` instances run other UNIX-like systems or on Windows.

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.

MongoDB preallocates its database files before using them and often creates large files. As such, you should use the Ext4 and XFS file systems:

- In general, if you use the Ext4 file system, use at least version 2.6.23 of the Linux Kernel.
- In general, if you use the XFS file system, use at least version 2.6.25 of the Linux Kernel.
- Some Linux distributions require different versions of the kernel to support using ext4 and/or xfs:

Linux Distribution	Filesystem	Kernel Version
CentOS 5.5	ext4, xfs	2.6.18-194.el5
CentOS 5.6	ext4, xfs	2.6.18-238.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	2.6.18-238
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

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

- 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 [UNIX ulimit Settings](#) (page 219). 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.
- Do not use hugepages virtual memory pages as MongoDB performs better with normal virtual memory pages.
- Disable NUMA in your BIOS. If that is not possible see [MongoDB on NUMA Hardware](#) (page 153).
- 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.
- Use the Network Time Protocol (NTP) to synchronize time among your hosts. This is especially important in sharded clusters.

Networking

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 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, `auth` is not enabled and `mongod` assumes a trusted environment. You can enable [security/auth](#) (page 235) mode if you need it.

See documents in the [Security](#) (page 233) section for additional information, specifically:

- [Configuration Options](#) (page 238)
- [Firewalls](#) (page 239)
- [Configure Linux iptables Firewall for MongoDB](#) (page 241)
- [Configure Windows netsh Firewall for MongoDB](#) (page 245)

For Windows users, consider the Windows Server Technet Article on TCP Configuration⁴¹ when deploying MongoDB on Windows.

MongoDB on Virtual Environments

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

EC2 MongoDB is compatible with EC2 and requires no configuration changes specific to the environment.

You may alternately choose to obtain a set of Amazon Machine Images (AMI) that bundle together MongoDB and Amazon's Provisioned IOPS storage volumes. Provisioned IOPS can greatly increase MongoDB's performance and ease of use. For more information, see [this blog post](#)⁴².

⁴¹<http://technet.microsoft.com/en-us/library/dd349797.aspx>

⁴²<http://www.mongodb.com/blog/post/provisioned-iops-aws-marketplace-significantly-boosts-mongodb-performance-ease-use>

VMWare MongoDB is compatible with VMWare. As some users have run into issues with VMWare's memory overcommit feature, disabling the feature is recommended.

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 consistent. If not using journaling, first stop mongod, then clone the VM, and finally, restart mongod.

OpenVZ Some users have had issues when running MongoDB on some older version of OpenVZ due to its handling of virtual memory, as with VMWare.

This issue seems to have been resolved in the more recent versions of OpenVZ.

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-ng⁴³ 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 *Backup Strategies for MongoDB Systems* (page 134).

4.1.2 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.

Data Center Awareness (page 157) 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 158) 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 160) 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.

⁴³<http://www.gropp.org/?id=projects&sub=bwm-ng>

Data Center Awareness

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 \(page 157\)](#) MongoDB lets you specify that certain application operations use certain `mongod` instances.

[Tag Aware Sharding \(page 547\)](#) Tags associate specific ranges of *shard key* values with specific shards for use in managing deployment patterns.

[Manage Shard Tags \(page 548\)](#) Use tags to associate specific ranges of shard key values with specific shards.

Operational Segregation in MongoDB Deployments

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* \(page 401\)](#), which controls how drivers help applications target read operations to members of a replica set.
- [*Write Concerns* \(page 44\)](#), which controls how MongoDB ensures that write operations propagate to members of a replica set.
- [*Replica Set Tags* \(page 446\)](#), 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* \(page 547\)](#), 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](#) (page 373), :and doc:`sharding` </sharding>.

Further Reading

- The [Write Concern](#) (page 44) and [Read Preference](#) (page 401) documents, which address capabilities related to data center awareness.
- [Deploy a Geographically Redundant Replica Set](#) (page 421).

Capped Collections

Capped collections are fixed-size collections that support high-throughput operations that insert, retrieve, and delete 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 `createCollection` for more information on creating capped collections.

Capped collections have the following behaviors:

- 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, they can support higher insertion throughput.
- Capped collections guarantee that insertion order is identical to the order on disk (*natural order*) and do so by prohibiting updates that increase document size. Capped collections only allow updates that fit the original document size, which ensures a document does not change its location on disk.
- 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.

Recommendations and Restrictions

- You *can* update documents in a collection after inserting them. *However*, these updates **cannot** cause the documents to grow. If the update operation causes the document to grow beyond their original size, the update operation will fail.

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

- You cannot delete documents from a capped collection. To remove all records from a capped collection, use the ‘`emptycapped`’ command. To remove the collection entirely, use the `drop()` method.
- You cannot shard a capped collection.

- Capped collections created after 2.2 have an `_id` field and an index on the `_id` field by default. Capped collections created before 2.2 do not have an index on the `_id` field by default. If you are using capped collections with replication prior to 2.2, you should explicitly create an index on the `_id` field.

Warning: If you have a capped collection in a *replica set* outside of the `local` database, before 2.2, you should create a unique index on `_id`. Ensure uniqueness using the `unique: true` option to the `ensureIndex()` method or by using an `ObjectId` for the `_id` field. Alternately, you can use the `autoIndexId` option to create when creating the capped collection, as in the [Query a Capped Collection](#) (page 159) procedure.

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

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 } )
```

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.

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 160). 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 160) 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 [Create Tailable Cursor](#) (page 71) for information on creating a tailable cursor.

Expire Data from Collections by Setting TTL

- [Enable TTL for a Collection](#) (page 160)
 - [Expire after a Certain Number of Seconds](#) (page 161)
 - [Expire at a Certain Clock Time](#) (page 161)
- [Constraints](#) (page 162)

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

Enable TTL for a Collection

To enable TTL for a collection, use the `ensureIndex()` method to create a TTL index, as shown in the examples below. MongoDB begins removing expired documents as soon as the index finishes building.

Note: When the TTL thread is active, you will see a [delete](#) (page 40) operations in the output of `db.currentOp()` or in the data collected by the [database profiler](#) (page 171).

Note: When enabling TTL on *replica sets*, the TTL background thread runs *only* on *primary* members. *Secondary* members replicate deletion operations from the primary.

Warning: The TTL index does not guarantee that expired data will be deleted immediately. There may be a delay between the time a document expires and the time that MongoDB removes the document from the database. The background task that removes expired documents runs *every 60 seconds*. As a result, documents may remain in a collection *after* they expire but *before* the background task runs or completes. The duration of the removal operation depends on the workload of your mongod instance. Therefore, expired data may exist for some time *beyond* the 60 second period between runs of the background task.

With the exception of the background thread, A TTL index supports queries in the same way normal indexes do. You can use TTL indexes to expire documents in one of two ways, either:

- remove documents a certain number of seconds after creation. The index will support queries for the creation time of the documents. Alternately,
- specify an explicit expiration time. The index will support queries for the expiration-time of the document.

Expire after a Certain Number of Seconds Begin by creating a TTL index and specify an expireAfterSeconds value of 3600. This sets the an expiration time of 1 hour after the time specified by the value of the indexed field. The following example, creates an index on the log.events collection's status field:

```
db.log.events.ensureIndex( { "status": 1 }, { expireAfterSeconds: 3600 } )
```

To expire documents a certain number of seconds after creation, give the date field a value corresponding to the insertion time of the documents. For example, given the index on the log.events collection with the expireAfterSeconds value of 0, and a current date of July 22, 2013: 13:00:00, consider the document in the following insert() operation:

```
db.log.events.insert( {
  "status": new Date('July 22, 2013: 13:00:00'),
  "logEvent": 2,
  "logMessage": "Success!",
} )
```

The status field *must* hold values of BSON date type or an array of BSON date-typed objects.

MongoDB will automatically delete documents from the log.events collection when at least one of the values of a document's status field is a time older than the number of seconds specified in expireAfterSeconds.

Expire at a Certain Clock Time Begin by creating a TTL index and specify an expireAfterSeconds value of 0. The following example, creates an index on the log.events collection's status field:

```
db.log.events.ensureIndex( { "status": 1 }, { expireAfterSeconds: 0 } )
```

To expire documents at a certain clock time, give the date field a value corresponding to the time a document should expire. For example, given the index on the log.events collection with the expireAfterSeconds value of 0, and a current date of July 22, 2013: 13:00:00, consider the document in the following insert() operation:

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

The status field *must* hold values of BSON date type or an array of BSON date-typed objects.

MongoDB will automatically delete documents from the `log.events` collection when at least one of the values of a document's `status` field is a time older than the number of seconds specified in `expireAfterSeconds`.

Constraints

- The `_id` field does not support TTL indexes.
- You cannot create a TTL index on a field that already has an index.
- A document will not expire if the indexed field does not exist.
- A document will not expire if the indexed field is not a date *BSON type* or an array of date *BSON types*.
- The TTL index may not be compound (may not have multiple fields).
- If the TTL field holds an array, and there are multiple date-typed data in the index, the document will expire when the *lowest* (i.e. earliest) date matches the expiration threshold.
- You cannot create a TTL index on a capped collection, because MongoDB cannot remove documents from a capped collection.
- You cannot use `ensureIndex()` to change the value of `expireAfterSeconds`. Instead use the `collMod` database command in conjunction with the `index` collection flag.

Important: All collections with an index using the `expireAfterSeconds` option have `usePowerOf2Sizes` enabled. Users cannot modify this setting. As a result of enabling `usePowerOf2Sizes`, MongoDB must allocate more disk space relative to data size. This approach helps mitigate the possibility of storage fragmentation caused by frequent delete operations and leads to more predictable storage use patterns.

4.1.3 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.

[**Evaluate Performance of Current Operations**](#) (page 162) MongoDB provides introspection tools that describe the query execution process, to allow users to test queries and build more efficient queries.

[**Use Capped Collections for Fast Writes and Reads**](#) (page 163) Outlines a use case for [*Capped Collections*](#) (page 158) to optimize certain data ingestion work flows.

[**Optimize Query Performance**](#) (page 163) Introduces the use of [*projections*](#) (page 30) to reduce the amount of data MongoDB must set to clients.

[**Design Notes**](#) (page 165) A collection of notes related to the architecture, design, and administration of MongoDB-based applications.

Evaluate Performance of Current Operations

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 141).

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 `explain()` method returns statistics on a query, and reports the index MongoDB selected to fulfill the query, as well as information about the internal operation of the query.

Example

To use `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()
```

Use Capped Collections for Fast Writes and Reads

Use Capped Collections for Fast Writes

[Capped Collections](#) (page 158) are circular, fixed-size collections that keep documents well-ordered, even without the use of an index. This means that capped collections can receive very high-speed writes and sequential reads.

These collections are particularly useful for keeping log files but are not limited to that purpose. Use capped collections where appropriate.

Use Natural Order for Fast Reads

To return documents in the order they exist on disk, return sorted operations using the `$natural` operator. On a capped collection, this also returns the documents in the order in which they were written.

Natural order does not use indexes but can be fast for operations when you want to select the first or last items on disk.

See also:

`sort()` and `limit()`.

Optimize Query Performance

Create Indexes to Support Queries

For commonly issued queries, create [indexes](#) (page 309). If a query searches multiple fields, create a [compound index](#) (page 318). 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.ensureIndex( { 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.ensureIndex( { timestamp : 1 } )
```

Optimizes this query:

```
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](#) (page 279).

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 *ursors* 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 : -1 } )
```

For more information on using projections, see [Limit Fields to Return from a Query](#) (page 61).

Use \$hint to Select a Particular Index

In most cases the [query optimizer](#) (page 35) 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.

Design Notes

This page details features of MongoDB that may be important to bear in mind when designing your 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 [Data Modeling Concepts](#) (page 97) 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 sub-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 `http://docs.mongodb.org/manual/`, and/or
- using `$toLower` or `$toUpper` in the [aggregation framework](#) (page 277).

Type Sensitive Fields MongoDB data is stored in the [BSON⁴⁴](#) format, a binary encoded serialization of JSON-like documents. BSON encodes additional type information. See [bsonspec.org⁴⁵](#) for more information.

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

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

⁴⁴<http://docs.mongodb.org/meta-driver/latest/legacy/bson/>

⁴⁵<http://bsonspec.org/#/specification>

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](#) (page 43).

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](#) (page 102).

No Fully Generalized Transactions MongoDB does not have *fully generalized transactions* (page 74). Creating rich documents that closely resemble and reflect your application-level objects to (words).

Replica Set Considerations

Use an Odd Number of Replica Set Members [Replica sets](#) (page 373) 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](#) (page 392). 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 136) for a detailed discussion of MongoDB's monitoring options.
2. Specify appropriate write concern.
3. If your application requires *manual* fail over, you can configure your secondaries as [priority 0](#) (page 382). 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](#) (page 397).

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 550).
- Consider [pre-splitting](#) (page 513) a sharded collection before a massive bulk import.

4.2 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 167) Describes routine management operations, including configuration and performance analysis.

Manage mongod Processes (page 168) Start, configure, and manage running `mongod` process.

Rotate Log Files (page 177) Archive the current log files and start new ones.

Backup and Recovery (page 184) Outlines procedures for data backup and restoration with `mongod` instances and deployments.

Backup and Restore with Filesystem Snapshots (page 188) 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 193) Detailed procedures and considerations for backing up sharded clusters and single shards.

Recover Data after an Unexpected Shutdown (page 200) Recover data from MongoDB data files that were not properly closed or are in an inconsistent state.

MongoDB Scripting (page 202) An introduction to the scripting capabilities of the `mongo` shell and the scripting capabilities embedded in MongoDB instances.

MongoDB Tutorials (page 181) A complete list of tutorials in the MongoDB Manual that address MongoDB operation and use.

4.2.1 Configuration, Maintenance, and Analysis

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

Use Database Commands (page 168) The process for running database commands that provide basic database operations.

Manage mongod Processes (page 168) Start, configure, and manage running `mongod` process.

Analyze Performance of Database Operations (page 171) Collect data that introspects the performance of query and update operations on a `mongod` instance.

Monitor MongoDB with SNMP (page 174) The SNMP extension, available in MongoDB Enterprise, allows MongoDB to report data into SNMP traps.

Rotate Log Files (page 177) Archive the current log files and start new ones.

Manage Journaling (page 178) 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 180) Describes how to store JavaScript functions on a MongoDB server.

MongoDB Tutorials (page 181) A complete list of tutorials in the MongoDB Manual that address MongoDB operation and use.

Use Database Commands

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 <http://docs.mongodb.org/manual/reference/command> for list of all commands sorted by function, and <http://docs.mongodb.org/manual/reference/command> for a list of all commands sorted alphabetically.

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* (page 92) 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.

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.

Manage mongod Processes

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 <http://docs.mongodb.org/manual/reference/program/mongod>. MongoDB can also run as a

Windows service. For details, see [MongoDB as a Windows Service](#) (page 14). To install MongoDB, see [Install MongoDB](#) (page 3).

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

See also:

[Run-time Database Configuration](#) (page 143), <http://docs.mongodb.org/manual/reference/program/mongod>,
<http://docs.mongodb.org/manual/reference/program/mongos>, and
<http://docs.mongodb.org/manual/reference/configuration-options>.

Start `mongod`

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](#) (page 234).

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 common configuration deployments, configurations for common use cases, see [Run-time Database Configuration](#) (page 143).

Stop mongod

To stop a mongod instance not running as a daemon, press `Control+C`. MongoDB stops when all ongoing operations are complete and does a clean exit, flushing and closing all data files.

To stop a mongod instance running in the background or foreground, issue the `db.shutdownServer()` helper in the mongo shell. Use the following sequence:

1. To open the mongo shell for a mongod instance running on the default port of 27017, issue the following command:

```
mongo
```

2. To switch to the admin database and shutdown the mongod instance, issue the following commands:

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

You may only use `db.shutdownServer()` when connected to the mongod when authenticated to the admin database or on systems without authentication connected via the localhost interface.

Alternately, you can shut down the mongod instance from a driver using the `shutdown` command. For details, see the [drivers documentation](#) (page 92) for your driver.

mongod Shutdown and Replica Sets If the mongod is the *primary* in a *replica set*, the shutdown process for these mongod instances 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.

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})
```

Sending a UNIX INT or TERM Signal

You can cleanly stop mongod using a SIGINT or SIGTERM signal on UNIX-like systems. Either ^C for a non-daemon mongod instance, kill -2 <pid>, or kill -15 <pid> will cleanly terminate the mongod instance.

Terminating a mongod instance that is **not** running with *journaling* with kill -9 <pid> (i.e. SIGKILL) will probably cause data corruption.

To recover data in situations where mongod instances have not terminated cleanly *without journaling* see *Recover Data after an Unexpected Shutdown* (page 200).

Analyze Performance of Database Operations

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 171) is also configurable when enabling profiling.

The database profiler writes all the data it collects to the `system.profile` (page 223) collection, which is a *capped collection* (page 158). See *Database Profiler Output* (page 226) for overview of the data in the `system.profile` (page 223) 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 226)
- `Profile` Command
- <http://docs.mongodb.org/manual/reference/method/db.currentOp>

Profiling Levels

The following profiling levels are available:

- 0 - the profiler is off, does not collect any data.
- 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 `slowms` runtime option or the `setParameter` command. See the *Specify the Threshold for Slow Operations* (page 172) 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* (page 92) if you want to control the profiler from within your application.

When you enable profiling, you also set the *profiling level* (page 171). The profiler records data in the `system.profile` (page 223) collection. MongoDB creates the `system.profile` (page 223) collection in a database after you enable profiling for that database.

To enable profiling and set the profiling level, issue 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 172) 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 171), 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 171).

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:

[profile](#) and [slowms](#).

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 223) collection.

To view profiling information, query the [system.profile](#) (page 223) collection. To view example queries, see [Profiler Overhead](#) (page 174).

For an explanation of the output data, see [Database Profiler Output](#) (page 226).

Example Profiler Data Queries This section displays example queries to the [system.profile](#) (page 223) collection. For an explanation of the query output, see [Database Profiler Output](#) (page 226).

To return the most recent 10 log entries in the [system.profile](#) (page 223) 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 223) 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.

To change the size of the `system.profile` (page 223) collection, you must:

1. Disable profiling.
2. Drop the `system.profile` (page 223) collection.
3. Create a new `system.profile` (page 223) collection.
4. Re-enable profiling.

For example, to create a new `system.profile` (page 223) 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)
```

Monitor MongoDB with SNMP

New in version 2.2.

Enterprise Feature

This feature is only available in MongoDB Enterprise.

This document outlines the use and operation of MongoDB's SNMP extension, which is only available in MongoDB Enterprise⁴⁶.

⁴⁶<http://www.mongodb.com/products/mongodb-enterprise>

Prerequisites

Install MongoDB Enterprise *MongoDB Enterprise*

Included Files The Enterprise packages contain the following files:

- MONGO-MIB.txt:

The MIB file that describes the data (i.e. schema) for MongoDB's SNMP output

- mongod.conf:

The SNMP configuration file for reading the SNMP output of MongoDB. The SNMP configures the community names, permissions, access controls, etc.

Required Packages To use SNMP, you must install several prerequisites. The names of the packages vary by distribution and are as follows:

- Ubuntu 11.04 requires libssl0.9.8, snmp-mibs-downloader, snmp, and snmpd. Issue a command such as the following to install these packages:

```
sudo apt-get install libssl0.9.8 snmp snmpd snmp-mibs-downloader
```

- Red Hat Enterprise Linux 6.x series and Amazon Linux AMI require libssl, net-snmp, net-snmp-libs, and net-snmp-utils. Issue a command such as the following to install these packages:

```
sudo yum install libssl net-snmp net-snmp-libs net-snmp-utils
```

- SUSE Enterprise Linux requires libopenssl0_9_8, libsnmp15, slessp1-libsnmp15, and snmp-mibs. Issue a command such as the following to install these packages:

```
sudo zypper install libopenssl0_9_8 libsnmp15 slessp1-libsnmp15 snmp-mibs
```

Configure SNMP

Install MIB Configuration Files Ensure that the MIB directory /usr/share/snmp/mibs exists. If not, issue the following command:

```
sudo mkdir -p /usr/share/snmp/mibs
```

Use the following command to create a symbolic link:

```
sudo ln -s <path>MONGO-MIB.txt /usr/share/snmp/mibs/
```

Replace [/path/to/mongodb/distribution/] with the path to your MONGO-MIB.txt configuration file.

Copy the mongod.conf file into the /etc/snmp directory with the following command:

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

Start Up You can control MongoDB Enterprise using default or custom control scripts, just as with any other mongod:

Use the following command to view all SNMP options available in your MongoDB:

```
mongod --help | grep snmp
```

The above command should return the following output:

Module snmp options:

```
--snmp-subagent      run snmp subagent  
--snmp-master       run snmp as master
```

Ensure that the following directories exist:

- /data/db/ (This is the path where MongoDB stores the data files.)
- /var/log/mongodb/ (This is the path where MongoDB writes the log output.)

If they do not, issue the following command:

```
mkdir -p /var/log/mongodb/ /data/db/
```

Start the **mongod** instance with the following command:

```
mongod --snmp-master --port 3001 --fork --dbpath /data/db/ --logpath /var/log/mongodb/1.log
```

Optionally, you can set these options in a configuration file.

To check if **mongod** is running with SNMP support, issue the following command:

```
ps -ef | grep 'mongod --snmp'
```

The command should return output that includes the following line. This indicates that the proper **mongod** instance is running:

```
systemuser 31415 10260 0 Jul13 pts/16 00:00:00 mongod --snmp-master --port 3001 # [...]
```

Test SNMP Check for the snmp agent process listening on port 1161 with the following command:

```
sudo lsof -i :1161
```

which return the following output:

```
COMMAND PID USER FD TYPE DEVICE SIZE/OFF NODE NAME  
mongod 9238 sysadmin 10u IPv4 96469 0t0 UDP localhost:health-polling
```

Similarly, this command:

```
netstat -anp | grep 1161
```

should return the following output:

```
udp 0 0 127.0.0.1:1161 0.0.0.0:* 9238/<path>/mongod
```

Run snmpwalk Locally `snmpwalk` provides tools for retrieving and parsing the SNMP data according to the MIB. If you installed all of the required packages above, your system will have `snmpwalk`.

Issue the following command to collect data from **mongod** using SNMP:

```
snmpwalk -m MONGO-MIB -v 2c -c mongodb 127.0.0.1:1161 1.3.6.1.4.1.37601
```

You may also choose to specify the path to the MIB file:

```
snmpwalk -m /usr/share/snmp/mibs/MONGO-MIB -v 2c -c mongodb 127.0.0.1:1161 1.3.6.1.4.1.37601
```

Use this command *only* to ensure that you can retrieve and validate SNMP data from MongoDB.

Troubleshooting

Always check the logs for errors if something does not run as expected; see the log at `/var/log/mongodb/1.log`. The presence of the following line indicates that the `mongod` cannot read the `/etc/snmp/mongod.conf` file:

```
[SNMPAgent] warning: error starting SNMPAgent as master err:1
```

Rotate Log Files

- [Overview \(page 177\)](#)
- [Log Rotation With MongoDB \(page 177\)](#)
- [Syslog Log Rotation \(page 178\)](#)

Overview

Log rotation using MongoDB's standard 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 (GMT) 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.

MongoDB's standard approach to log rotation only rotates logs in response to the `logRotate` command, or when the `mongod` or `mongos` process receives a `SIGUSR1` signal from the operating system.

Alternately, you may configure `mongod` to send log data to `syslog`. In this case, you can take advantage of alternate logrotation tools.

See also:

For information on logging, see the [Process Logging \(page 139\)](#) section.

Log Rotation With MongoDB

The following steps create and rotate a log file:

1. Start a `mongod` with verbose logging, with appending enabled, and with the following log file:

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

2. In a separate terminal, list the matching files:

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

For results, you get:

```
server1.log
```

3. Rotate the log file using *one* of the following methods.

- From the `mongo` shell, issue the `logRotate` command from the `admin` database:

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

This is the only available method to rotate log files on Windows systems.

- For Linux systems, rotate logs for a single process by issuing the following command:

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

4. List the matching files again:

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

For results you get something similar to the following. The timestamps will be different.

```
server1.log  server1.log.2011-11-24T23-30-00
```

The example results indicate a log rotation performed at exactly 11:30 pm on November 24th, 2011 UTC, which is the local time offset by the local time zone. The original log file is the one with the timestamp. The new log is `server1.log` file.

If you issue a second `logRotate` command an hour later, then an additional file would appear when listing matching files, as in the following example:

```
server1.log  server1.log.2011-11-24T23-30-00  server1.log.2011-11-25T00-30-00
```

This operation does not modify the `server1.log.2011-11-24T23-30-00` file created earlier, while `server1.log.2011-11-25T00-30-00` is the previous `server1.log` file, renamed. `server1.log` is a new, empty file that receives all new log output.

Syslog Log Rotation

New in version 2.2.

To configure mongod to send log data to syslog rather than writing log data to a file, use the following procedure.

1. Start a mongod with the `syslog` option.
2. Store and rotate the log output using your system's default log rotation mechanism.

Important: You cannot use `syslog` with `logpath`.

Manage Journaling

MongoDB uses *write ahead logging* to an on-disk *journal* to guarantee [write operation](#) (page 40) durability and to provide crash resiliency. 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.

Without a journal, if `mongod` exits unexpectedly, you must assume your data is in an inconsistent state, and you must run either [repair](#) (page 200) or, preferably, [resync](#) (page 445) from a clean member of the replica set.

With journaling enabled, if `mongod` stops unexpectedly, the program can recover everything written to the journal, and the data remains in 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 `journalCommitInterval` for more information on the default.

With journaling, if you want a data set to reside entirely in RAM, you need enough RAM to hold the data set plus the “write working set.” The “write working set” is the amount of unique data you expect to see written between re-mappings of the private view. For information on views, see [Storage Views used in Journaling](#) (page 229).

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

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.

If no journal files exist, when `mongod` starts, it must preallocate new journal files. During this operation, the `mongod` is not listening for connections until preallocation completes: for some systems this may take a several minutes. During this period your applications and the `mongo` shell are not available.

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 200).

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

Get Commit Acknowledgment You can get commit acknowledgment with the `getLastError` command and the `j` option. For details, see [Write Concern Reference](#) (page 80).

Avoid Preallocation Lag To avoid [preallocation lag](#) (page 229), 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 starting 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:

```
web admin interface listening on port 11000
```

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 Changed in version 2.0.

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.

Store a JavaScript Function on the Server

Note: We do **not** recommend using server-side stored functions if possible.

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 example:

```
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. `eval` command or the mongo shell method `db.eval()`, `$where` operator, `mapReduce` or mongo shell method `db.collection.mapReduce()`).

Consider the following example from the mongo shell that first saves a function named `echoFunction` to the `system.js` collection and calls the function using `db.eval()` method:

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

db.eval( "echoFunction( 'test' )" )
```

See <http://github.com/mongodb/mongo/tree/master/jstests/storefunc.js> for a full example.

New in version 2.1: 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);
```

MongoDB Tutorials

This page lists the tutorials available as part of the MongoDB Manual. In addition to these documents, you can refer to the introductory [MongoDB Tutorial](#) (page 18). If there is a process or pattern that you would like to see included here, please open a [Jira Case](#)⁴⁷.

Getting Started

- [Install MongoDB on Linux Systems](#) (page 9)
- [Install MongoDB on Red Hat Enterprise, CentOS, or Fedora](#) (page 4)
- [Install MongoDB on Debian](#) (page 7)
- [Install MongoDB on Ubuntu](#) (page 6)
- [Install MongoDB on OS X](#) (page 10)
- [Install MongoDB on Windows](#) (page 12)
- [Getting Started with MongoDB](#) (page 18)
- [Generate Test Data](#) (page 23)

Administration

Replica Sets

- [Deploy a Replica Set](#) (page 416)

⁴⁷<https://jira.mongodb.org/browse/DOCS>

- [*Convert a Standalone to a Replica Set* \(page 428\)](#)
- [*Add Members to a Replica Set* \(page 429\)](#)
- [*Remove Members from Replica Set* \(page 431\)](#)
- [*Replace a Replica Set Member* \(page 433\)](#)
- [*Adjust Priority for Replica Set Member* \(page 434\)](#)
- [*Resync a Member of a Replica Set* \(page 445\)](#)
- [*Deploy a Geographically Redundant Replica Set* \(page 421\)](#)
- [*Change the Size of the Oplog* \(page 441\)](#)
- [*Force a Member to Become Primary* \(page 443\)](#)
- [*Change Hostnames in a Replica Set* \(page 453\)](#)
- [*Add an Arbiter to Replica Set* \(page 427\)](#)
- [*Convert a Secondary to an Arbiter* \(page 438\)](#)
- [*Configure a Secondary's Sync Target* \(page 457\)](#)
- [*Configure a Delayed Replica Set Member* \(page 437\)](#)
- [*Configure a Hidden Replica Set Member* \(page 436\)](#)
- [*Configure Non-Voting Replica Set Member* \(page 438\)](#)
- [*Prevent Secondary from Becoming Primary* \(page 434\)](#)
- [*Configure Replica Set Tag Sets* \(page 446\)](#)
- [*Manage Chained Replication* \(page 452\)](#)
- [*Reconfigure a Replica Set with Unavailable Members* \(page 450\)](#)
- [*Recover Data after an Unexpected Shutdown* \(page 200\)](#)
- [*Troubleshoot Replica Sets* \(page 457\)](#)

Sharding

- [*Deploy a Sharded Cluster* \(page 514\)](#)
- [*Convert a Replica Set to a Replicated Sharded Cluster* \(page 523\)](#)
- [*Add Shards to a Cluster* \(page 521\)](#)
- [*Remove Shards from an Existing Sharded Cluster* \(page 541\)](#)
- [*Deploy Three Config Servers for Production Deployments* \(page 522\)](#)
- [*Migrate Config Servers with the Same Hostname* \(page 531\)](#)
- [*Migrate Config Servers with Different Hostnames* \(page 532\)](#)
- [*Replace a Config Server* \(page 533\)](#)
- [*Migrate a Sharded Cluster to Different Hardware* \(page 533\)](#)
- [*Backup Cluster Metadata* \(page 536\)](#)
- [*Backup a Small Sharded Cluster with mongodump* \(page 193\)](#)
- [*Backup a Sharded Cluster with Filesystem Snapshots* \(page 194\)](#)

- [Backup a Sharded Cluster with Database Dumps](#) (page 195)
- [Restore a Single Shard](#) (page 196)
- [Restore a Sharded Cluster](#) (page 197)
- [Schedule Backup Window for Sharded Clusters](#) (page 196)
- [Manage Shard Tags](#) (page 548)

Basic Operations

- [Use Database Commands](#) (page 168)
- [Recover Data after an Unexpected Shutdown](#) (page 200)
- [Copy Databases Between Instances](#) (page 198)
- [Expire Data from Collections by Setting TTL](#) (page 160)
- [Analyze Performance of Database Operations](#) (page 171)
- [Rotate Log Files](#) (page 177)
- [Build Old Style Indexes](#) (page 341)
- [Manage mongod Processes](#) (page 168)
- [Backup and Restore with MongoDB Tools](#) (page 185)
- [Backup and Restore with Filesystem Snapshots](#) (page 188)

Security

- [Configure Linux iptables Firewall for MongoDB](#) (page 241)
- [Configure Windows netsh Firewall for MongoDB](#) (page 245)
- [Enable Authentication](#) (page 253)
- [Create a User Administrator](#) (page 254)
- [Add a User to a Database](#) (page 255)
- [Generate a Key File](#) (page 256)
- [Deploy MongoDB with Kerberos Authentication](#) (page 257)
- [Create a Vulnerability Report](#) (page 261)

Development Patterns

- [Perform Two Phase Commits](#) (page 66)
- [Isolate Sequence of Operations](#) (page 74)
- [Create an Auto-Incrementing Sequence Field](#) (page 75)
- [Enforce Unique Keys for Sharded Collections](#) (page 550)
- [Aggregation Examples](#) (page 288)
- [Model Data to Support Keyword Search](#) (page 118)
- [Limit Number of Elements in an Array after an Update](#) (page 78)

- [Perform Incremental Map-Reduce](#) (page 298)
- [Troubleshoot the Map Function](#) (page 300)
- [Troubleshoot the Reduce Function](#) (page 301)
- [Store a JavaScript Function on the Server](#) (page 180)

Text Search Patterns

- [Enable Text Search](#) (page 354)
- [Create a text Index](#) (page 355)
- [Search String Content for Text](#) (page 355)
- [Specify a Language for Text Index](#) (page 358)
- [Create text Index with Long Name](#) (page 360)
- [Control Search Results with Weights](#) (page 360)
- [Create text Index to Cover Queries](#) (page 363)
- [Limit the Number of Entries Scanned](#) (page 361)

Data Modeling Patterns

- [Model One-to-One Relationships with Embedded Documents](#) (page 104)
- [Model One-to-Many Relationships with Embedded Documents](#) (page 105)
- [Model One-to-Many Relationships with Document References](#) (page 106)
- [Model Data for Atomic Operations](#) (page 117)
- [Model Tree Structures with Parent References](#) (page 110)
- [Model Tree Structures with Child References](#) (page 111)
- [Model Tree Structures with Materialized Paths](#) (page 114)
- [Model Tree Structures with Nested Sets](#) (page 116)

4.2.2 Backup and Recovery

The following tutorials describe backup and restoration for a `mongod` instance:

Backup and Restore with MongoDB Tools ([page 185](#)) The procedure for writing the contents of a database to a BSON (i.e. binary) dump file for backing up MongoDB databases.

Backup and Restore with Filesystem Snapshots ([page 188](#)) 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 191](#)) Describes procedure for restoring a replica set from an archived backup such as a `mongodump` or [MMS Backup](#)⁴⁸ file.

Backup and Restore Sharded Clusters ([page 193](#)) Detailed procedures and considerations for backing up sharded clusters and single shards.

⁴⁸<http://mms.mongodb.com>

[Copy Databases Between Instances](#) (page 198) Copy databases between mongod instances or within a single mongod instance or deployment.

[Recover Data after an Unexpected Shutdown](#) (page 200) Recover data from MongoDB data files that were not properly closed or are in an inconsistent state.

Backup and Restore with MongoDB Tools

This document describes the process for writing the entire contents of your MongoDB instance to a file in a binary format. If disk-level snapshots are not available, this approach provides the best option for full system database backups. If your system has disk level snapshot capabilities, consider the backup methods described in [Backup and Restore with Filesystem Snapshots](#) (page 188).

See also:

[Backup Strategies for MongoDB Systems](#) (page 134), <http://docs.mongodb.org/manual/reference/program/mongodump> and <http://docs.mongodb.org/manual/reference/program/mongorestore>.

Backup a Database with `mongodump`

Important: `mongodump` does *not* create output for the local database.

Basic `mongodump` Operations The `mongodump` utility can back up data by either:

- connecting to a running `mongod` or `mongos` instance, or
- accessing data files without an active 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
```

Note: Data created by the `mongodump` tool from the 2.2 distribution is incompatible with versions of `mongorestore` from the 2.0 distribution and earlier.

To limit the amount of data included in the database dump, you can specify `--db` and `--collection` as options to the `mongodump` command. For example:

```
mongodump --dbpath /data/db/ --out /data/backup/
```

```
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.

```
mongodump --collection collection --db test
```

This command creates a dump of the collection named `collection` from the database `test` in a `dump/` subdirectory of the current working directory.

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 of the backup procedure to until the backup procedure completes. This backup procedure, in conjunction with `mongorestore --oplogReplay`, allows you to restore a backup that reflects a consistent and specific moment in time.

Create Backups Without a Running `mongod` Instance If your MongoDB instance is not running, you can use the `--dbpath` option to specify the location to your MongoDB instance's database files. `mongodump` reads from the data files directly with this operation. This locks the data directory to prevent conflicting writes. The `mongod` process must *not* be running or attached to these data files when you run `mongodump` in this configuration. Consider the following example:

Example

Backup a MongoDB Instance Without a Running `mongod`

Given a MongoDB instance that contains the `customers`, `products`, and `suppliers` databases, the following `mongodump` operation backs up the databases using the `--dbpath` option, which specifies the location of the database files on the host:

```
mongodump --dbpath /data -o dataout
```

The `--out` option allows you to specify the directory where `mongodump` will save the backup. `mongodump` creates a separate backup directory for each of the backed up databases: `dataout/customers`, `dataout/products`, and `dataout/suppliers`.

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/backup/
```

On any `mongodump` command you may, as above, specify username and password credentials to specify database authentication.

Restore a Database with `mongorestore`

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 can restore data either by:

- connecting to a running `mongod` or `mongos` directly, or
- writing to a set of MongoDB data files without use of a running `mongod`.

`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>
```

To use `mongorestore` to write to data files without using a running `mongod`, use a command with the following prototype form:

```
mongorestore --dbpath <database path> <path to the backup>
```

Consider the following example:

```
mongorestore dump-2012-10-25/
```

Here, `mongorestore` imports the database backup in the `dump-2012-10-25` directory to the `mongod` instance running on the localhost interface.

Restore Point in Time Olog 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 a Subset of data from a Binary Database Dump `mongorestore` also includes the ability to filter to all input before inserting it into the new database. Consider the following example:

```
mongorestore --filter '{"field": 1}'
```

Here, `mongorestore` only adds documents to the database from the dump located in the `dump/` folder if the documents have a field name `field` that holds a value of 1. Enclose the filter in single quotes (e.g. `'`) to prevent the filter from interacting with your shell environment.

Restore Without a Running `mongod` `mongorestore` can write data to MongoDB data files without needing to connect to a `mongod` directly.

Example

Restore a Database Without a Running `mongod`

Given a set of backed up databases in the `/data/backup/` directory:

- `/data/backup/customers`,
- `/data/backup/products`, and
- `/data/backup/suppliers`

The following `mongorestore` command restores the `products` database. The command uses the `--dbpath` option to specify the path to the MongoDB data files:

```
mongorestore --dbpath /data/db --journal /data/backup/products
```

The `mongorestore` imports the database backup in the `/data/backup/products` directory to the `mongod` instance that runs on the localhost interface. The `mongorestore` operation imports the backup even if the `mongod` is not running.

The `--journal` option ensures that `mongorestore` records all operation in the durability *journal*. The journal prevents data file corruption if anything (e.g. power failure, disk failure, etc.) interrupts the restore operation.

See also:

<http://docs.mongodb.org/manual/reference/program/mongodump>
<http://docs.mongodb.org/manual/reference/program/mongorestore>.

and

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/mon
```

As above, you may specify username and password connections if your `mongod` requires authentication.

Backup and Restore with Filesystem Snapshots

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 more system configuration outside of MongoDB.

See also:

[Backup Strategies for MongoDB Systems](#) (page 134) and [Backup and Restore with MongoDB Tools](#) (page 185).

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.

Snapshots have the following limitations:

- The database must be in a consistent or recoverable state 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. If writes are *in progress* when the backup occurs, the data files will reflect an inconsistent state. With *journaling* 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.

- 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.

- Ensure that you copy data from snapshots and onto other systems to ensure that data is safe from site failures.
- 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 that do not have Journaling Enabled](#) (page 191).
- 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 189).

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 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 <http://docs.mongodb.org/manual/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 <http://docs.mongodb.org/manual/dev/vg0/mongodb> and the creation of the snapshot (i.e. <http://docs.mongodb.org/manual/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 move the data to separate storage. Your system might try to compress the backup images as you move the offline. The following procedure fully archives the data from the snapshot:

```
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 `http://docs.mongodb.org/manual/dev/vg0/mdb-snap01` device is not 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 `http://docs.mongodb.org/manual/dev/vg0` volume group. The path to the new device will be `http://docs.mongodb.org/manual/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 `journal` 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 190) 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 that do not have Journaling Enabled

If your `mongod` instance does not run with journaling enabled, or if your journal is 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*).

1. To flush writes to disk and to “lock” the database (to prevent further writes), issue the `db.fsyncLock()` method in the `mongo` shell:


```
db.fsyncLock();
```
2. Perform the backup operation described in [Create a Snapshot](#) (page 189).
3. To unlock the database after the snapshot has completed, use the following command in the `mongo` shell:


```
db.fsyncUnlock();
```

Note: Changed in version 2.0: MongoDB 2.0 added `db.fsyncLock()` and `db.fsyncUnlock()` helpers to the `mongo` shell. Prior to this version, use the `fsync` command with the `lock` option, as follows:

```
db.runCommand( { fsync: 1, lock: true } );
db.runCommand( { fsync: 1, lock: false } );
```

Note: The database cannot be locked with `db.fsyncLock()` while profiling is enabled. You must disable profiling before locking the database with `db.fsyncLock()`. Disable profiling using `db.setProfilingLevel()` as follows in the `mongo` shell:

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

Warning: Changed in version 2.2: When used in combination with `fsync` or `db.fsyncLock()`, `mongod` may block some reads, including those from `mongodump`, when queued write operation waits behind the `fsync` lock.

Restore a Replica Set from MongoDB Backups

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* simply 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

1. Obtain backup MongoDB Database files. These files may come from a *file system snapshot* (page 188). The MMS Backup Service produces MongoDB database files for stored snapshots⁴⁹ and point and time snapshots⁵⁰.

You can also use `mongorestore` to restore database files using data created with `mongodump`. See *Backup and Restore with MongoDB Tools* (page 185) for more information.

2. Start a `mongod` using data files from the backup as the `dbpath`. In the following example, `/data/db` is the `dbpath` to the data files:

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

3. Convert your standalone `mongod` process to a single node replica set by shutting down the `mongod` instance, and restarting it with the `--rep1Set` option, as in the following example:

```
mongod --dbpath /data/db --rep1Set <rep1Name>
```

Optional

Consider explicitly setting a `oplogSize` to control the size of the *oplog* created for this replica set member.

4. Connect to the `mongod` instance.
5. Use `rs.initiate()` to initiate the new replica set.

Add Members to the Replica Set

MongoDB provides two options for restoring secondary members of a replica set:

1. Manually copy the database files to each data directory.
2. Allow *initial sync* (page 408) 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.

1. Shut down the `mongod` instance that you restored. Using `--shutdown` or `db.shutdownServer()` to ensure a clean shut down.
2. Copy the *primary*’s data directory into the `dbpath` of the other members of the replica set. The `dbpath` is `/data/db` by default.
3. Start the `mongod` instance that you restored.
4. In a mongo shell connected to the *primary*, add the *secondaries* to the replica set using `rs.add()`. See *Deploy a Replica Set* (page 416) for more information about deploying a replica set.

⁴⁹<http://mms.mongodb.org/help/backup/tutorial/restore-snapshot>

⁵⁰<http://mms.mongodb.org/help/backup/tutorial/restore-from-point-in-time-snapshot>

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.

1. Ensure that the data directories on the prospective replica set members are empty.
2. Add each prospective member to the replica set. [Initial Sync](#) (page 408) will copy the data from the *primary* to the other members of the replica set.

Backup and Restore Sharded Clusters

The following tutorials describe backup and restoration for sharded clusters:

[Backup a Small Sharded Cluster with mongodump](#) (page 193) 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 194) 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 195) Create backups using `mongodump` to back up each component in the cluster individually.

[Schedule Backup Window for Sharded Clusters](#) (page 196) Limit the operation of the cluster balancer to provide a window for regular backup operations.

[Restore a Single Shard](#) (page 196) An outline of the procedure and consideration for restoring a single shard from a backup.

[Restore a Sharded Cluster](#) (page 197) An outline of the procedure and consideration for restoring an *entire* sharded cluster from backup.

Backup a Small Sharded Cluster with `mongodump`

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.

Read [Sharded Cluster Backup Considerations](#) (page 135) for a high-level overview of important considerations as well as a list of alternate backup tutorials.

Important: By default `mongodump` issue its queries to the non-primary nodes.

Procedure

Capture Data

Note: If you use `mongodump` without specifying a database or collection, `mongodump` will capture collection data and the cluster meta-data from the [config servers](#) (page 496).

You cannot use the `--oplog` option for `mongodump` when capturing data from `mongos`. This option is only available when running directly against a *replica set* member.

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 514) for more information.

Backup a Sharded Cluster with Filesystem Snapshots

Overview 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 that uses `mongodump` to create binary database dumps when file-system snapshots are not available. See [Backup a Sharded Cluster with Database Dumps](#) (page 195) for the alternate procedure.

See [Sharded Cluster Backup Considerations](#) (page 135) for a full higher level overview backing up a sharded cluster as well as links to other tutorials that provide alternate procedures.

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.

Procedure 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.

1. 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, and see the [Disable the Balancer](#) (page 540) procedure.

Warning: It is essential that you stop the balancer before creating backups. If the balancer remains active, your resulting backups could have duplicate data or miss some data, as *chunks* may migrate while recording backups.

2. 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, you must:

- use the `db.fsyncLock()` method in the `mongo` shell connected to a single secondary member of the replica set that provides shard `mongod` instance.
 - Shutdown one of the [config servers](#) (page 496), to prevent all metadata changes during the backup process.
3. Use `mongodump` to backup one of the [config servers](#) (page 496). This backs up the cluster's metadata. You only need to back up one config server, as they all hold the same data.

Issue this command against one of the config `mongod` instances or via the `mongos`:

```
mongodump --db config
```

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 procedures in [Backup and Restore with Filesystem Snapshots](#) (page 188).
5. Unlock all locked replica set members of each shard using the `db.fsyncUnlock()` method in the mongo shell.
6. 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)
```

Backup a Sharded Cluster with Database Dumps

Overview 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 a Sharded Cluster with Filesystem Snapshots](#) (page 194).

See [Sharded Cluster Backup Considerations](#) (page 135) for a full higher level overview of backing up a sharded cluster as well as links to other tutorials that provide alternate procedures.

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.

Procedure 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 mongodump to capture the backup data. 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 of 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.

1. 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, and see the [Disable the Balancer](#) (page 540) procedure.

Warning: It is essential that you stop the balancer before creating backups. If the balancer remains active, your resulting backups could have duplicate data or miss some data, as *chunks* migrate while recording backups.

2. 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, you must:

- Shutdown one member of each replica set.

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 [Oplog Size](#) (page 406) for more information.

- Shutdown one of the *config servers* (page 496), to prevent all metadata changes during the backup process.

3. Use `mongodump` to backup one of the *config servers* (page 496). This backs up the cluster's metadata. You only need to back up one config server, as they all hold the same data.

Issue this command against one of the config `mongod` instances or via the `mongos`:

```
mongodump --journal --db config
```

4. Back up the replica set members of the shards that shut down using `mongodump` and specifying the `--dbpath` option. You may back up the shards in parallel. Consider the following invocation:

```
mongodump --journal --dbpath /data/db/ --out /data/backup/
```

You must run this command on the system where the `mongod` ran. This operation will use journaling and create a dump of the entire `mongod` instance with data files stored in `/data/db/`. `mongodump` will write the output of this dump to the `/data/backup/` directory.

5. Restart all stopped replica set members of each shard as normal and allow them to catch up with the state of the primary.
6. 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)
```

Schedule Backup Window for Sharded Clusters

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 540) 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.

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 : "23:00" } } }
```

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

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:

- [Sharded Cluster Backup Considerations](#) (page 135)

- [Restore a Sharded Cluster](#) (page 197)
- [Backup Strategies for MongoDB Systems](#) (page 134)

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.

1. Restore the shard as you would any other `mongod` instance. See [Backup Strategies for MongoDB Systems](#) (page 134) for overviews of these procedures.
2. 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.
3. 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](#) (page 555).

Restore a Sharded Cluster

Overview The procedure outlined in this document addresses how to restore an entire sharded cluster. For information on related backup procedures consider the following tutorials which describe backup procedures in greater detail:

- [Backup a Sharded Cluster with Filesystem Snapshots](#) (page 194)
- [Backup a Sharded Cluster with Database Dumps](#) (page 195)

The exact procedure used to restore a database depends on the method used to capture the backup. See the [Backup Strategies for MongoDB Systems](#) (page 134) document for an overview of backups with MongoDB, as well as [Sharded Cluster Backup Considerations](#) (page 135) which provides an overview of the high level concepts important for backing up sharded clusters.

Procedure

1. Stop all `mongod` and `mongos` processes.
2. If shard hostnames have changed, you must manually update the `shards` collection in the [Config Database](#) (page 555) to use the new hostnames. Do the following:
 - (a) Start the three [config servers](#) (page 496) by issuing commands similar to the following, using values appropriate to your configuration:


```
mongod --configsvr --dbpath /data/configdb --port 27019
```
 - (b) Restore the [Config Database](#) (page 555) on each config server.
 - (c) Start one `mongos` instance.
 - (d) Update the [Config Database](#) (page 555) collection named `shards` to reflect the new hostnames.
3. Restore the following:
 - Data files for each server in each *shard*. Because replica sets provide each production shard, restore all the members of the replica set or use the other standard approaches for restoring a replica set from backup. See the [Restore a Snapshot](#) (page 190) and [Restore a Database with mongorestore](#) (page 186) sections for details on these procedures.
 - Data files for each [config server](#) (page 496), if you have not already done so in the previous step.

4. Restart all the mongos instances.
5. Restart all the mongod instances.
6. 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
```

Copy Databases Between Instances

Synopsis

MongoDB provides the `copydb` and `clone` *database commands* to support migrations of entire logical databases between `mongod` instances. With these commands you can copy data between instances with a simple interface without the need for an intermediate stage. The `db.cloneDatabase()` and `db.copyDatabase()` provide helpers for these operations in the `mongo` shell. These migration helpers run the commands on the **destination** server and *pull* data from the **source** server..

Data migrations that require an intermediate stage or that involve more than one database instance are beyond the scope of this tutorial. `copydb` and `clone` are more ideal for use cases that resemble the following use cases:

- data migrations,
- data warehousing, and
- seeding test environments.

Also consider the [Backup Strategies for MongoDB Systems](#) (page 134) and [Import and Export MongoDB Data](#) (page 147) documentation for more related information.

Note: `copydb` and `clone` do not produce point-in-time snapshots of the source database. Write traffic to the source or destination database during the copy process will result divergent data sets.

Considerations

- You must run `copydb` or `clone` on the destination server.
- You cannot use `copydb` or `clone` with databases that have a sharded collection in a *sharded cluster*, or any database via a `mongos`.
- You *can* use `copydb` or `clone` with databases that do not have sharded collections in a *cluster* when you're connected directly to the `mongod` instance.
- You can run `copydb` or `clone` commands on a *secondary* member of a replica set, with properly configured *read preference*.
- Each destination `mongod` instance must have enough free disk space on the destination server for the database you are copying. Use the `db.stats()` operation to check the size of the database on the source `mongod` instance. For more information, see `db.stats()`.

Processes

Copy and Rename a Database To copy a database from one MongoDB instance to another and rename the database in the process, use the `copydb` command, or the `db.copyDatabase()` helper in the `mongo` shell.

Use the following procedure to copy the database named `test` on server `db0.example.net` to the server named `db1.example.net` and rename it to `records` in the process:

- Verify that the database, `test` exists on the source `mongod` instance running on the `db0.example.net` host.
- Connect to the destination server, running on the `db1.example.net` host, using the `mongo` shell.
- Model your operation on the following command:

```
db.copyDatabase( "test", "records", "db0.example.net" )
```

Rename a Database You can also use `copydb` or the `db.copyDatabase()` helper to:

- rename a database within a single MongoDB instance or
- create a duplicate database for testing purposes.

Use the following procedure to rename the `test` database `records` on a single `mongod` instance:

- Connect to the `mongod` using the `mongo` shell.
- Model your operation on the following command:

```
db.copyDatabase( "test", "records" )
```

Copy a Database with Authentication To copy a database from a source MongoDB instance that has authentication enabled, you can specify authentication credentials to the `copydb` command or the `db.copyDatabase()` helper in the `mongo` shell.

In the following operation, you will copy the `test` database from the `mongod` running on `db0.example.net` to the `records` database on the local instance (e.g. `db1.example.net`.) Because the `mongod` instance running on `db0.example.net` requires authentication for all connections, you will need to pass `db.copyDatabase()` authentication credentials, as in the following procedure:

- Connect to the destination `mongod` instance running on the `db1.example.net` host using the `mongo` shell.
- Issue the following command:

```
db.copyDatabase( "test", "records", db0.example.net, "<username>", "<password>" )
```

Replace `<username>` and `<password>` with your authentication credentials.

Clone a Database The `clone` command copies a database between `mongod` instances like `copydb`; however, `clone` preserves the database name from the source instance on the destination `mongod`.

For many operations, `clone` is functionally equivalent to `copydb`, but it has a more simple syntax and a more narrow use. The `mongo` shell provides the `db.cloneDatabase()` helper as a wrapper around `clone`.

You can use the following procedure to clone a database from the `mongod` instance running on `db0.example.net` to the `mongod` running on `db1.example.net`:

- Connect to the destination `mongod` instance running on the `db1.example.net` host using the `mongo` shell.
- Issue the following command to specify the name of the database you want to copy:

```
use records
```

- Use the following operation to initiate the `clone` operation:

```
db.cloneDatabase( "db0.example.net" )
```

Recover Data after an Unexpected Shutdown

If MongoDB does not shutdown cleanly ⁵¹ the on-disk representation of the data files will likely reflect an inconsistent state which could lead to data corruption. ⁵²

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.

See also:

The *Administration* (page 133) documents, including *Replica Set Syncing* (page 406), and the documentation on the repair, repairpath, and journal 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* (page 406) 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 mongod.lock file exists in your dbpath, or the optional --repairpath, you will see a message that contains the following line:

```
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 134) 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 445).

There are two processes to repair data files that result from an unexpected shutdown:

⁵¹ To ensure a clean shut down, use the db.shutdownServer() from the mongo shell, your control script, the mongod --shutdown option on Linux systems, “Control-C” when running mongod in interactive mode, or kill \$(pidof mongod) or kill -2 \$(pidof mongod).

⁵² 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.

1. 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. This does not modify or alter the existing data files.

You do not need to remove the `mongod.lock` file before using this procedure.

2. 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 To repair your data files using the `--repairpath` option to preserve the original data files unmodified:

1. Start `mongod` using `--repair` to read the existing data files.

```
mongod --dbpath /data/db --repair --repairpath /data/db0
```

When this completes, the new repaired data files will be in the `/data/db0` directory.

2. 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 data files in the `/data/db` directory.

To repair your data files without preserving the original files, do not use the `--repairpath` option, as in the following procedure:

1. Remove the stale lock file:

```
rm /data/db/mongod.lock
```

Replace `/data/db` with your `dbpath` where your MongoDB instance's data files reside.

Warning: After you remove the `mongod.lock` file you *must* run the `--repair` process before using your database.

2. Start `mongod` using `--repair` to read the existing data files.

```
mongod --dbpath /data/db --repair
```

When this completes, the repaired data files will replace the original data files in the `/data/db` directory.

3. 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 134) 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 445).

4.2.3 MongoDB Scripting

The mongo shell is an interactive JavaScript shell for MongoDB, and is part of all [MongoDB distributions](#)⁵³. This section provides an introduction to the shell, and outlines key functions, operations, and use of the mongo shell. Also consider [FAQ: The mongo Shell](#) (page 584) and the `shell` method and other relevant reference material.

Note: Most examples in the MongoDB Manual use the mongo shell; however, many *drivers* (page 92) provide similar interfaces to MongoDB.

[Server-side JavaScript \(page 202\)](#) Details MongoDB's support for executing JavaScript code for server-side operations.

[Data Types in the mongo Shell \(page 203\)](#) Describes the super-set of JSON available for use in the mongo shell.

[Write Scripts for the mongo Shell \(page 206\)](#) An introduction to the mongo shell for writing scripts to manipulate data and administer MongoDB.

[Getting Started with the mongo Shell \(page 208\)](#) Introduces the use and operation of the MongoDB shell.

[Access the mongo Shell Help Information \(page 212\)](#) Describes the available methods for accessing online help for the operation of the mongo interactive shell.

[mongo Shell Quick Reference \(page 214\)](#) A high level reference to the use and operation of the mongo shell.

Server-side JavaScript

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.

Overview

MongoDB supports the execution of JavaScript code for the following server-side operations:

- `mapReduce` and the corresponding mongo shell method `db.collection.mapReduce()`. See [Map-Reduce](#) (page 280) for more information.
- `eval` command, and the corresponding mongo shell method `db.eval()`
- `$where` operator
- [Running .js files via a mongo shell Instance on the Server](#) (page 203)

JavaScript in MongoDB

Although the above operations use JavaScript, most interactions with MongoDB do not use JavaScript but use an *idiomatic driver* (page 92) in the language of the interacting application.

See also:

[Store a JavaScript Function on the Server](#) (page 180)

⁵³<http://www.mongodb.org/downloads>

You can disable all server-side execution of JavaScript, by passing the `--noscripting` option on the command line or setting `noscripting` in a configuration file.

Running .js files via a mongo shell Instance on the Server

You can run a JavaScript (.js) file using a mongo shell instance 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 214) 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
<code>show dbs, show databases</code>	<code>db.adminCommand('listDatabases')</code>
<code>use <db></code>	<code>db = db.getSiblingDB('<db>')</code>
<code>show collections</code>	<code>db.getCollectionNames()</code>
<code>show users</code>	<code>db.system.users.find()</code>
<code>show log <logname></code>	<code>db.adminCommand({ 'getLog' : '<logname>' })</code>
<code>show logs</code>	<code>db.adminCommand({ 'getLog' : '*' })</code>
<code>it</code>	<pre>cursor = db.collection.find() if (cursor.hasNext()){ cursor.next(); }</pre>

Concurrency

Refer to the individual method or operator documentation for any concurrency information. See also the [concurrency table](#) (page 588).

Data Types in the mongo Shell

MongoDB *BSON* provide support for additional data types than *JSON*. [Drivers](#) (page 92) 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 [MongoDB Extended JSON](#) (page 223) for additional information.

Types

Date The mongo shell provides various options to return the date, either as a string or as an object:

- `Date()` method which returns the current date as a string.
- `Date()` constructor which returns an `ISODate` object when used with the `new` operator.

- `ISODate()` constructor which returns an `ISODate` object when used with `or` without the `new` operator.

Consider the following examples:

- 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`.

- To get the date as an `ISODate` object, instantiate a new instance using the `Date()` constructor with the `new` operator, as in the following example:

```
var myDateObject = new Date();
```

- To print the value of the variable, type the variable name in the shell, as in the following:

```
myDateObject
```

The result is the value of `myDateObject`:

```
ISODate("2012-12-19T06:01:17.171Z")
```

- To verify the type, use the `typeof` operator, as in the following:

```
typeof myDateObject
```

The operation returns `object`.

- To get the date as an `ISODate` object, instantiate a new instance using the `ISODate()` constructor *without* the `new` operator, as in the following example:

```
var myDateObject2 = 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:

```
myDateObject2
```

The result is the value of `myDateObject2`:

```
ISODate("2012-12-19T06:15:33.035Z")
```

- To verify the type, use the `typeof` operator, as in the following:

```
typeof myDateObject2
```

The operation returns `object`.

ObjectId The mongo shell provides the `ObjectId()` wrapper class around `ObjectId` data types. To generate a new ObjectId, use the following operation in the mongo shell:

```
new ObjectId
```

See

[ObjectId](#) (page 127) 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()` class to handle 64-bit integers.

The `NumberLong()` constructor accepts the long as a string:

```
NumberLong("2090845886852")
```

The following examples use the `NumberLong()` class 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 following operators:

- `instanceof` returns a boolean to test if a value has a specific type.
- `typeof` returns the type of a field.

Example

Consider the following operations using `instanceof` and `typeof`:

- The following operation tests whether the `_id` field is of type `ObjectId`:

```
mydoc._id instanceof ObjectId
```

The operation returns `true`.

- 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.

Write Scripts for the mongo Shell

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 202), and see the [Running .js files via a mongo shell Instance on the Server](#) (page 203) 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");
```

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");
```

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`.
- Inside the script, call `db.getLastError()` explicitly to wait for the result of [write operations](#) (page 40).

- 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
<code>show dbs, show databases</code>	<code>db.adminCommand('listDatabases')</code>
<code>use <db></code>	<code>db = db.getSiblingDB('<db>')</code>
<code>show collections</code>	<code>db.getCollectionNames()</code>
<code>show users</code>	<code>db.system.users.find()</code>
<code>show log <logname></code>	<code>db.adminCommand({ 'getLog' : '<logname>' })</code>
<code>show logs</code>	<code>db.adminCommand({ 'getLog' : '*' })</code>
<code>it</code>	<pre>cursor = db.collection.find() if (cursor.hasNext()){ cursor.next(); }</pre>

- 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 206) 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.

Getting Started with the `mongo` Shell

This document provides a basic introduction to using the `mongo` shell. See [Install MongoDB](#) (page 3) 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 [How can I access different databases temporarily?](#) (page 585) 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.

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 213) 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 [Cursors](#) (page 33).

See also [Cursor Help](#) (page 213) 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](#) (page 18)
- [mongo Shell Quick Reference](#) (page 214)
- [Read Operations](#) (page 29)
- [Write Operations](#) (page 40)
- [Indexing Tutorials](#) (page 334)
- [Read Operations](#) (page 29)
- [Write Operations](#) (page 40)

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

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.

Access the mongo Shell Help Information

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 202), and
- [mongo Shell Quick Reference](#) (page 214).

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.addUser()`:

```
db.addUser
```

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](#) (page 29) 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](#) (page 33). 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
```

mongo Shell Quick Reference

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 executable can be started with numerous options. See [mongo executable](#) page for details on all available options.

The following table displays some common options for mongo:

Op-tion	Description
<code>--help</code>	Show command line options
<code>--nodb</code>	Start mongo shell without connecting to a database. To connect later, see Opening New Connections (page 206).
<code>--shell</code>	Used in conjunction with a JavaScript file (i.e. <code><file.js></code>) to continue in the mongo shell after running the JavaScript file. See JavaScript file (page 207) 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
<code>help</code>	Show help.
<code>db.help()</code>	Show help for database methods.
<code>db.<collection>.help</code>	Show help on collection methods. The <collection> can be the name of an existing collection or a non-existing collection.
<code>show dbs</code>	Print a list of all databases on the server.
<code>use <db></code>	Switch current database to <db>. The mongo shell variable db is set to the current database.
<code>show collections</code>	Print a list of all collections for current database
<code>show users</code>	Print a list of users for current database.
<code>show profile</code>	Print the five most recent operations that took 1 millisecond or more. See documentation on the database profiler (page 171) for more information.
<code>show databases</code>	New in version 2.4: Print a list of all available databases.
<code>load()</code>	Execute a JavaScript file. See Getting Started with the mongo Shell (page 208) for more information.

Basic Shell JavaScript Operations

The mongo shell provides numerous <http://docs.mongodb.org/manual/reference/method> methods 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
<code>db.auth()</code>	If running in secure mode, authenticate the user.
<code>coll = db.<collection></code>	Set a specific collection in the current database to a variable <code>coll</code> , as in the following example: <code>coll = db.myCollection;</code> You can perform operations on the <code>myCollection</code> using the variable, as in the following example: <code>coll.find();</code>
<code>find()</code>	Find all documents in the collection and returns a cursor. See the <code>db.collection.find()</code> and Query Documents (page 57) for more information and examples. See Cursors (page 33) for additional information on cursor handling in the <code>mongo</code> shell.
<code>insert()</code>	Insert a new document into the collection.
<code>update()</code>	Update an existing document in the collection. See Write Operations (page 40) for more information.
<code>save()</code>	Insert either a new document or update an existing document in the collection. See Write Operations (page 40) for more information.
<code>remove()</code>	Delete documents from the collection. See Write Operations (page 40) for more information.
<code>drop()</code>	Drops or removes completely the collection.
<code>ensureIndex()</code>	Create a new index on the collection if the index does not exist; otherwise, the operation has no effect.
<code>db.getSiblingDB()</code>	Return a reference to another database using this same connection without explicitly switching the current database. This allows for cross database queries. See How can I access different databases temporarily? (page 585) for more information.

For more information on performing operations in the shell, see:

- [MongoDB CRUD Concepts](#) (page 27)
- [Read Operations](#) (page 29)
- [Write Operations](#) (page 40)
- <http://docs.mongodb.org/manual/reference/method>

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
Continued on next page	

Table 4.1 – continued from previous page

Keystroke	Function
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
<code>db.collection.find(<query>)</code>	<p>Find the documents matching the <code><query></code> criteria in the collection. If the <code><query></code> criteria is not specified or is empty (i.e <code>{ } </code>), the read operation selects all documents in the collection.</p> <p>The following example selects the documents in the <code>users</code> collection with the <code>name</code> field equal to "Joe":</p> <pre>coll = db.users; coll.find({ name: "Joe" });</pre> <p>For more information on specifying the <code><query></code> criteria, see Query Documents (page 57).</p>
<code>db.collection.find(<query>, <projection>)</code>	<p>Find documents matching the <code><query></code> criteria and return just specific fields in the <code><projection></code>.</p> <p>The following example selects all documents from the collection but returns only the <code>name</code> field and the <code>_id</code> field. The <code>_id</code> is always returned unless explicitly specified to not return.</p> <pre>coll = db.users; coll.find({ }, { name: true });</pre> <p>For more information on specifying the <code><projection></code>, see Limit Fields to Return from a Query (page 61).</p>
<code>db.collection.find().sort(<sort order>)</code>	<p>Return results in the specified <code><sort order></code>.</p> <p>The following example selects all documents from the collection and returns the results sorted by the <code>name</code> field in ascending order (1). Use -1 for descending order:</p> <pre>coll = db.users; coll.find().sort({ name: 1 });</pre>
<code>db.collection.find(<query>).sort(<sort order>)</code> <code>db.collection.find(...).limit(<n>)</code>	<p>Return the documents matching the <code><query></code> criteria in the specified <code><sort order></code>.</p> <p>Limit result to <code><n></code> rows. Highly recommended if you need only a certain number of rows for best performance.</p>
<code>db.collection.find(...).skip(<n>)</code> <code>count()</code> <code>db.collection.find(<query>).count()</code>	<p>Skip <code><n></code> results.</p> <p>Returns total number of documents in the collection.</p> <p>Returns the total number of documents that match the query.</p> <p>The <code>count()</code> ignores <code>limit()</code> and <code>skip()</code>. For example, if 100 records match but the limit is 10, <code>count()</code> will return 100. This will be faster than iterating yourself, but still take time.</p>
<code>db.collection.findOne(<query>)</code>	<p>Find and return a single document. Returns null if not found.</p> <p>The following example selects a single document in the <code>users</code> collection with the <code>name</code> field matches to "Joe":</p> <pre>coll = db.users; coll.findOne({ name: "Joe" });</pre> <p>Internally, the <code>findOne()</code> method is the <code>find()</code> method with a <code>limit(1)</code>.</p>

See [Query Documents](#) (page 57) and [Read Operations](#) (page 29) documentation for more information and examples. See <http://docs.mongodb.org/manual/reference/operator> to specify other query operators.

Error Checking Methods

The mongo shell provides numerous *administrative database methods*, including error checking methods. These methods are:

Error Checking Methods	Description
<code>db.getLastError()</code>	Returns error message from the last operation.
<code>db.getLastErrorObj()</code>	Returns the error document from the last operation.

Administrative Command Helpers

The following table lists some common methods to support database administration:

JavaScript Database Administration Methods	Description
<code>db.cloneDatabase(<host>)</code>	Clone the current database from the <code><host></code> specified. The <code><host></code> database instance must be in noauth mode.
<code>db.copyDatabase(<from>, <to>, <host>)</code>	Copy the <code><from></code> database from the <code><host></code> to the <code><to></code> database on the current server. The <code><host></code> database instance must be in noauth mode.
<code>db.fromColl.renameCollection(<coll>, <newColl>)</code>	<code>Rename collection from</code> <code>fromColl</code> to <code><toColl></code> .
<code>db.repairDatabase()</code>	Repair and compact the current database. This operation can be very slow on large databases.
<code>db.addUser(<user>, <pwd>)</code>	Add user to current database.
<code>db.getCollectionNames()</code>	Get the list of all collections in the current database.
<code>db.dropDatabase()</code>	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
<code>db = connect ("<host><:port>/<dbname>")</code>	Open a new database connection.
<code>conn = new Mongo() db = conn.getDB("dbname")</code>	Open a connection to a new server using <code>new Mongo()</code> . Use <code>getDB()</code> method of the connection to select a database.

See also [Opening New Connections](#) (page 206) for more information on the opening new connections from the mongo shell.

Miscellaneous

The following table displays some miscellaneous methods:

Method	Description
<code>Object.bsonsize(<document>)</code>	Prints the <i> BSON</i> size of an <document>

See the [MongoDB JavaScript API Documentation](#)⁵⁴ for a full list of JavaScript methods.

Additional Resources

Consider the following reference material that addresses the `mongo` shell and its interface:

- <http://docs.mongodb.org/manual/reference/program/mongo>
- <http://docs.mongodb.org/manual/reference/method>
- <http://docs.mongodb.org/manual/reference/operator>
- <http://docs.mongodb.org/manual/reference/command>
- [Aggregation Reference](#) (page 304)

Additionally, the MongoDB source code repository includes a `jstests` directory⁵⁵ which contains numerous `mongo` shell scripts.

See also:

The MongoDB Manual contains administrative documentation and tutorials though out several sections. See [Replica Set Tutorials](#) (page 415) and [Sharded Cluster Tutorials](#) (page 513) for additional tutorials and information.

4.3 Administration Reference

UNIX ulimit Settings (page 219) Describes user resources limits (i.e. `ulimit`) and introduces the considerations and optimal configurations for systems that run MongoDB deployments.

System Collections (page 223) Introduces the internal collections that MongoDB uses to track per-database metadata, including indexes, collections, and authentication credentials.

MongoDB Extended JSON (page 223) Describes the `JSON` super set used to express `BSON` documents in the `mongo` shell and other MongoDB tools.

Database Profiler Output (page 226) Describes the data collected by MongoDB's operation profiler, which introspects operations and reports data for analysis on performance and behavior.

Journaling Mechanics (page 229) 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 230) Lists the unique codes returned by `mongos` and `mongod` processes upon exit.

4.3.1 UNIX ulimit Settings

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` set-

⁵⁴<http://api.mongodb.org/js/index.html>

⁵⁵<https://github.com/mongodb/mongo/tree/master/jstests/>

tings. Edit the `soft nproc` and `hard nproc` values in the `/etc/security/limits.d/90-nproc.conf` file to increase the process limit.

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 `journal` 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 160), 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 `maxConns` 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: You cannot set `maxConns` to a value higher than 20000.

Review and Set Resource Limits

`ulimit`

Note: Both the “hard” and the “soft” `ulimit` 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.

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>
```

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.

Note: After changing the ulimit settings, you *must* restart the process to take advantage of the modified settings. You can use the <http://docs.mongodb.org/manualproc> 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.

/proc File System

Note: This section applies only to Linux operating systems.

The <http://docs.mongodb.org/manualproc> 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 $@; do
        cat /proc/$process/limits
    done
}
```

```
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 of the following invocations:

```
return-limits mongod  
return-limits mongos  
return-limits mongod mongos
```

The output of the first command may resemble the following:

```
[mongod #6809 -- limits]  
Limit          Soft Limit      Hard Limit      Units  
Max cpu time  unlimited      unlimited      seconds  
Max file size unlimited      unlimited      bytes  
Max data size unlimited      unlimited      bytes  
Max stack size 8720000      unlimited      bytes  
Max core file size 0          unlimited      bytes  
Max resident set unlimited      unlimited      bytes  
Max processes  192276        192276        processes  
Max open files 1024          4096         files  
Max locked memory 40960000    40960000    bytes  
Max address space unlimited      unlimited      bytes  
Max file locks  unlimited      unlimited      locks  
Max pending signals 192276    192276        signals  
Max msgqueue size 819200      819200        bytes  
Max nice priority 30           30            us  
Max realtime priority 65           65            us  
Max realtime timeout unlimited      unlimited      us
```

Recommended 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):** unlimited⁵⁶
- **-n (open files):** 64000
- **-m (memory size):** unlimited¹

⁵⁶ If you limit virtual or resident memory size on a system running MongoDB the operating system will refuse to honor additional allocation requests.

- `-u` (processes/threads): 32000

Always remember to restart your `mongod` and `mongos` instances after changing the `ulimit` settings to make sure that the settings change takes effect.

4.3.2 System Collections

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](#) (page 479), specifically for replication purposes.

Collections

System collections include these collections stored directly in the database:

`<database>.system.namespaces`

The [`<database>.system.namespaces`](#) (page 223) collection contains information about all of the database's collections. Additional namespace metadata exists in the `database.ns` files and is opaque to database users.

`<database>.system.indexes`

The [`<database>.system.indexes`](#) (page 223) collection lists all the indexes in the database. Add and remove data from this collection via the `ensureIndex()` and `dropIndex()`

`<database>.system.profile`

The [`<database>.system.profile`](#) (page 223) collection stores database profiling information. For information on profiling, see [Database Profiling](#) (page 141).

`<database>.system.users`

The [`<database>.system.users`](#) (page 268) collection stores credentials for users who have access to the database. For more information on this collection, see [Add a User to a Database](#) (page 255) and [`<database>.system.users`](#) (page 268).

`<database>.system.js`

The [`<database>.system.js`](#) (page 223) collection holds special JavaScript code for use in [server side JavaScript](#) (page 202). See [Store a JavaScript Function on the Server](#) (page 180) for more information.

4.3.3 MongoDB Extended JSON

MongoDB [import and export utilities](#) (page 147) (i.e. `mongoimport` and `mongoexport`) and MongoDB REST Interfaces⁵⁷ render an approximation of MongoDB `BSON` documents in JSON format.

The REST interface supports three different modes for document output:

- *Strict* mode that produces output that conforms to the [JSON RFC specifications](#)⁵⁸.
- *JavaScript* mode that produces output that most JavaScript interpreters can process (via the `--jsonp` option)
- *mongo Shell* mode produces output that the `mongo` shell can process. This is “extended” JavaScript format.

⁵⁷<http://docs.mongodb.org/ecosystem/tools/http-interfaces>

⁵⁸<http://www.json.org>

MongoDB can process of these representations in REST input.

Special representations of *BSON data* in JSON format make it possible to render information that have no obvious corresponding JSON. In some cases MongoDB supports multiple equivalent representations of the same type information. Consider the following table:

BSON Data Type	Strict Mode	JavaScript Mode (via JSONP)	mongo Shell Mode	Notes
<code>data_binary</code>	{ " <code>\$binary</code> ": "<bindata>", " <code>\$type</code> ": "<t>" }	{ " <code>\$binary</code> ": "<bindata>", " <code>\$type</code> ": "<t>" }	BinData(<t>, <bindata>)	<bindata> is the base64 representation of a binary string. <t> is the hexadecimal representation of a single byte that indicates the data type.
<code>data_date</code>	{ " <code>\$date</code> ": <date> }	<code>new Date(<date>)</code>	<code>new Date(<date>)</code>	<date> is the JSON representation of a 64-bit signed integer for milliseconds since epoch UTC (unsigned before version 1.9.1).
<code>data_timestamp</code>	{ " <code>\$timestamp</code> ": { " <code>t</code> ": <t>, " <code>i</code> ": <i> } }	{ " <code>\$timestamp</code> ": { " <code>t</code> ": <t>, " <code>i</code> ": <i> } }	Timestamp(<t>, <i>)	<t> is the JSON representation of a 32-bit unsigned integer for seconds since epoch. <i> is a 32-bit unsigned integer for the increment.
<code>data_regex</code>	{ " <code>\$regex</code> ": "<sRegex>," " <code>\$options</code> ": "<sOptions>" }	/<jRegex>/<jOptions>/<jRegex>/<jOptions>	<sRegex> is a string of valid JSON characters. <jRegex> is a string that may contain valid JSON characters and unescaped double quote ("") characters, but may not contain unescaped forward slash (http://docs.mongodb.org) characters. <sOptions> is a string containing the regex options represented by the letters of the alphabet. <jOptions> is a string that may contain only the characters 'g', 'i', 'm' and 's' (added in v1.9). Because the JavaScript and mongo Shell representations support a limited range of options, any non-conforming options will be dropped when converting to this representation. 225	
4.3. Administration Reference				
<code>data_oid</code>	{ " <code>\$oid</code> ": "<id>" }	{ " <code>\$oid</code> ": "<id>" }	<code>ObjectId("<id>")</code>	<id> is a 24-character hexadecimal string.

4.3.4 Database Profiler Output

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 171) section.

The database profiler writes data in the `system.profile` (page 223) collection, which is a *capped collection*. To view the profiler's output, use normal MongoDB queries on the `system.profile` (page 223) collection.

Note: Because the database profiler writes data to the `system.profile` (page 223) collection in a database, the profiler will profile some write activity, even for databases that are otherwise read-only.

Example `system.profile` Document

The documents in the `system.profile` (page 223) collection have the following form. This example document reflects an update operation:

```
{  
    "ts" : ISODate("2012-12-10T19:31:28.977Z"),  
    "op" : "update",  
    "ns" : "social.users",  
    "query" : {  
        "name" : "jane"  
    },  
    "updateobj" : {  
        "$set" : {  
            "likes" : [  
                "basketball",  
                "trekking"  
            ]  
        }  
    },  
    "nscanned" : 8,  
    "moved" : true,  
    "nmoved" : 1,  
    "nupdated" : 1,  
    "keyUpdates" : 0,  
    "numYield" : 0,  
    "lockStats" : {  
        "timeLockedMicros" : {  
            "r" : NumberLong(0),  
            "w" : NumberLong(258)  
        },  
        "timeAcquiringMicros" : {  
            "r" : NumberLong(0),  
            "w" : NumberLong(7)  
        }  
    },  
    "millis" : 0,  
    "client" : "127.0.0.1",  
    "user" : ""  
}
```

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.

`system.profile.ts`

The timestamp of the operation.

`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* (page 57) used.

`system.profile.command`

The command operation.

`system.profile.updateobj`

The <update> document passed in during an *update* (page 40) operation.

`system.profile.cursorid`

The ID of the cursor accessed by a getmore operation.

`system.profile.ntoreturn`

Changed in version 2.2: In 2.0, MongoDB includes this field for query and command operations. In 2.2, this information MongoDB also includes this field for getmore operations.

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 227) value would be 1. The `limit(5)` command would return five documents so the `ntoreturn` (page 227) value would be 5.

If the `ntoreturn` (page 227) 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`

New in version 2.2.

The number of documents the `skip()` method specified to skip.

`system.profile.nscanned`

The number of documents that MongoDB scans in the *index* (page 309) in order to carry out the operation.

In general, if `nscanned` (page 227) is much higher than `nreturned` (page 228), the database is scanning many objects to find the target objects. Consider creating an index to improve this.

`system.profile.moved`

If `moved` (page 227) has a value of `true` indicates that the update operation moved one or more documents

to a new location on disk. These operations take more time than in-place updates, and typically occur when documents grow as a result of document growth.

system.profile.nmoved

New in version 2.2.

The number of documents moved on disk by the operation.

system.profile.nupdated

New in version 2.2.

The number of documents updated by the operation.

system.profile.keyUpdates

New in version 2.2.

The number of [index](#) (page 309) 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.numYield

New in version 2.2.

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](#) (page 587).

system.profile.lockStats

New in version 2.2.

The time in microseconds the operation spent acquiring and holding locks. This field reports data for the following lock types:

- R - global read lock
- W - global write lock
- r - database-specific read lock
- w - database-specific write lock

system.profile.lockStats.timeLockedMicros

The time in microseconds the operation held a specific lock. For operations that require more than one lock, like those that lock the `local` database to update the `oplog`, then this value may be longer than the total length of the operation (i.e. [millis](#) (page 229).)

system.profile.lockStats.timeAcquiringMicros

The time in microseconds the operation spent waiting to acquire a specific lock.

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 228) can affect performance. To limit the size of the result document for a query operation, you can use any of the following:

- [Projections](#) (page 61)
- The `limit()` method
- The `batchSize()` method

Note: When MongoDB writes query profile information to the log, the [responseLength](#) (page 228) value is in a field named `reslen`.

system.profile.millis

The time in milliseconds for the server to perform the operation. This time does not include network time nor time to acquire the lock.

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.user

The authenticated user who ran the operation.

4.3.5 Journaling Mechanics

When running with journaling, MongoDB stores and applies [write operations](#) (page 40) in memory and in the journal before the changes are in the data files. This document discusses the implementation and mechanics of Journaling in MongoDB systems, see [Manage Journaling](#) (page 178) for information on configuring, tuning and managing journaling.

Journal Files

With journaling enabled, MongoDB creates a journal directory within the directory defined by `dbpath`, which is `/data/db` by default. The journal directory holds journal files, which contain write-ahead redo logs. The directory also holds a last-sequence-number file. A clean shutdown removes all the files in the journal directory.

Journal files are append-only files and have file names prefixed with `j._`. When a journal file holds 1 gigabyte of data, MongoDB creates a new journal file. Once MongoDB applies all the write operations in the journal files, it deletes these files. Unless you write *many* bytes of data per-second, the journal directory should contain only two or three journal files.

To limit the size of each journal file to 128 megabytes, use the `smallfiles` run time option when starting `mongod`.

To speed the frequent sequential writes that occur to the current journal file, you can ensure that the journal directory is on a different system.

Important: If you place the journal on a different filesystem from your data files you *cannot* use a filesystem snapshot to capture consistent backups of a `dbpath` directory.

Note: Depending on your file system, you might experience a preallocation lag the first time you start a `mongod` instance with journaling enabled.

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. The amount of time required to pre-allocate lag might last several minutes, during which 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](#) (page 179).

Storage Views used in Journaling

Journaling adds three storage views to MongoDB.

The `shared` view stores modified data for upload to the MongoDB data files. The `shared` view is the only view with direct access to the MongoDB data files. When running with journaling, `mongod` asks the operating system to

map your existing on-disk data files to the shared view memory view. The operating system maps the files but does not load them. MongoDB later loads data files to shared view as needed.

The private view stores data for use in [read operations](#) (page 29). MongoDB maps private view to the shared view and is the first place MongoDB applies new [write operations](#) (page 40).

The journal is an on-disk view that stores new write operations after MongoDB applies the operation to the private cache but before applying them to the data files. The journal provides durability. 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 upload to the data files.

How Journaling Records Write Operations

MongoDB copies the write operations to the journal in batches called group commits. See `journalCommitInterval` for more information on the default commit interval. These “group commits” help minimize the performance impact of journaling.

Journaling stores raw operations that allow MongoDB to reconstruct the following:

- document insertion/updates
- index modifications
- changes to the namespace files

As [write operations](#) (page 40) occur, MongoDB writes the data to the private view in RAM and then copies the write operations in batches to the journal. The journal stores the operations on disk to ensure durability. MongoDB adds the operations as entries on the journal’s forward pointer. Each entry describes which bytes the write operation changed in the data files.

MongoDB next applies the journal’s write operations to the shared view. At this point, the shared view becomes inconsistent with the data files.

At default intervals of 60 seconds, MongoDB asks the operating system to flush the shared view to disk. This brings the data files up-to-date with the latest write operations.

When MongoDB flushes write operations to the data files, MongoDB removes the write operations from the journal’s behind pointer. The behind pointer is always far back from the advanced pointer.

As part of journaling, MongoDB routinely asks the operating system to remap the shared view to the private view, for consistency.

Note: The interaction between the shared view and the on-disk data files is similar to how MongoDB works *without* journaling, which is that MongoDB asks the operating system to flush in-memory changes back to the data files every 60 seconds.

4.3.6 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` (page 481) 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 WSASStartup 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.

Security

This section outlines basic security and risk management strategies and access control. The included tutorials outline specific tasks for configuring firewalls, authentication, and system privileges.

Security Introduction ([page 233](#)) A high-level introduction to security and MongoDB deployments.

Security Concepts ([page 235](#)) The core documentation of security.

Access Control ([page 235](#)) Control access to MongoDB instances using authentication and authorization.

Network Exposure and Security ([page 238](#)) Discusses potential security risks related to the network and strategies for decreasing possible network-based attack vectors for MongoDB.

Security and MongoDB API Interfaces ([page 239](#)) Discusses potential risks related to MongoDB's JavaScript, HTTP and REST interfaces, including strategies to control those risks.

Sharded Cluster Security ([page 237](#)) MongoDB controls access to sharded clusters with key files.

Security Tutorials ([page 240](#)) Tutorials for enabling and configuring security features for MongoDB.

Create a Vulnerability Report ([page 261](#)) Report a vulnerability in MongoDB.

Network Security Tutorials ([page 241](#)) Ensure that the underlying network configuration supports a secure operating environment for MongoDB deployments, and appropriately limits access to MongoDB deployments.

Access Control Tutorials ([page 253](#)) MongoDB's access control system provides role-based access control for limiting access to MongoDB deployments. These tutorials describe procedures relevant for the operation and maintenance of this access control system.

Security Reference ([page 263](#)) Reference for security related functions.

5.1 Security Introduction

As with all software running in a networked environment, administrators of MongoDB must consider security and risk exposures for a MongoDB deployment. There are no magic solutions for risk mitigation, and maintaining a secure MongoDB deployment is an ongoing process.

5.1.1 Defense in Depth

The documents in this section takes a *Defense in Depth* approach to securing MongoDB deployments and addresses a number of different methods for managing risk and reducing risk exposure.

The intent of a *Defense In Depth* approach is to ensure there are no exploitable points of failure in your deployment that could allow an intruder or un-trusted party to access the data stored in the MongoDB database. The easiest and most effective way to reduce the risk of exploitation is to run MongoDB in a trusted environment, limit access, follow a system of least privilege, and follow best development and deployment practices.

5.1.2 Trusted Environments

The most effective way to reduce risk for MongoDB deployments is to run your entire MongoDB deployment, including all MongoDB components (i.e. mongod, mongos and application instances) in a *trusted environment*. Trusted environments use the following strategies to control access:

- Use network filter (e.g. firewall) rules that block all connections from unknown systems to MongoDB components.
- Bind mongod and mongos instances to specific IP addresses to limit accessibility.
- Limit MongoDB programs to non-public local networks, and virtual private networks.

5.1.3 Operational Practices to Reduce Risk

You may further reduce risk by [controlling access](#) (page 235) to the database by employing authentication and authorization. Require [authentication](#) (page 235) for access to MongoDB instances and require strong, complex, single purpose authentication credentials. This should be part of your internal security policy. Employ [authorization](#) (page 236) and deploy a model of least privilege, where all users have *only* the amount of access they need to accomplish required tasks and no more. See [Access Control](#) (page 235) for more information.

Follow the best application development and deployment practices, which includes: validating all inputs, managing sessions, and application-level access control.

Always run the mongod or mongos process as a *unique* user with the minimum required permissions and access. Never run a MongoDB program as a root or administrative users. The system users that run the MongoDB processes should have robust authentication credentials that prevent unauthorized or casual access.

To further limit the environment, you can run the mongod or mongos process in a chroot environment. Both user-based access restrictions and chroot configuration follow recommended conventions for administering all daemon processes on Unix-like systems.

5.1.4 Data Encryption

To support audit requirements, you may need to encrypt data stored in MongoDB. For best results, you can encrypt this data in the application layer by encrypting the content of fields that hold secure data.

Additionally, MongoDB¹ has a partnership² with Gazzang³ to encrypt and secure sensitive data within MongoDB. The solution encrypts data in real time and Gazzang provides advanced key management that ensures only authorized processes can access this data. The Gazzang software ensures that the cryptographic keys remain safe and ensures compliance with standards including HIPAA, PCI-DSS, and FERPA.

For more information, refer to the following resources: [Datasheet⁴](#) and [Webinar⁵](#).

¹<http://mongodb.com>

²<https://www.mongodb.com/partners/technology/gazzang>

³<http://www.gazzang.com/>

⁴<http://www.gazzang.com/images/datasheet-zNcrypt-for-MongoDB.pdf>

⁵<http://gazzang.com/resources/videos/partner-videos/item/209-gazzang-zncrypt-on-mongodb>

5.1.5 Additional Security Strategies

MongoDB provides various strategies to reduce network risk, such as configuring MongoDB or configuring firewalls for MongoDB. See [Network Exposure and Security](#) (page 238) for more information.

In addition, consider the strategies listed in [Security and MongoDB API Interfaces](#) (page 239) to reduce interface-related risks for the `mongo` shell, HTTP status interface and the REST API.

MongoDB Enterprise supports authentication using Kerberos. See [Deploy MongoDB with Kerberos Authentication](#) (page 257).

5.1.6 Vulnerability Notification

MongoDB takes security very seriously. If you discover a vulnerability in MongoDB, or would like to know more about our vulnerability reporting and response process, see the [Create a Vulnerability Report](#) (page 261) document.

5.2 Security Concepts

These documents introduce and address concepts and strategies related to security practices in MongoDB deployments.

[Access Control](#) (page 235) Control access to MongoDB instances using authentication and authorization.

[Inter-Process Authentication](#) (page 236) Components of a MongoDB sharded cluster or replica set deployment must be able to authenticate to each other to perform routine internal operations.

[Sharded Cluster Security](#) (page 237) MongoDB controls access to sharded clusters with key files.

[Network Exposure and Security](#) (page 238) Discusses potential security risks related to the network and strategies for decreasing possible network-based attack vectors for MongoDB.

[Security and MongoDB API Interfaces](#) (page 239) Discusses potential risks related to MongoDB's JavaScript, HTTP and REST interfaces, including strategies to control those risks.

5.2.1 Access Control

MongoDB provides support for authentication and authorization on a per-database level. Users exist in the context of a single logical database.

Authentication

MongoDB provisions authentication, or verification of the user identity, on a per-database level. Authentication disables anonymous access to the database. For basic authentication, MongoDB stores the user credentials in a database's `system.users` (page 268) collection.

Authentication is **disabled** by default. To enable authentication for a given `mongod` or `mongos` instance, use the `auth` and `keyFile` configuration settings. For details, see [Enable Authentication](#) (page 253).

For MongoDB Enterprise installations, authentication using a Kerberos service is available. See [Deploy MongoDB with Kerberos Authentication](#) (page 257).

Important: You can authenticate as only one user for a given database. If you authenticate to a database as one user and later authenticate on the same database as a different user, the second authentication invalidates the first. You can, however, log into a *different* database as a different user and not invalidate your authentication on other databases.

Authorization

MongoDB provisions authorization, or access to databases and operations, on a per-database level. MongoDB uses a role-based approach to authorization, storing each user's roles in a [privilege document](#) (page 263) in a database's `system.users` (page 268) collection. For more information on privilege documents and available user roles, see `system.users` [Privilege Documents](#) (page 268) and [User Privilege Roles in MongoDB](#) (page 263).

Important: The `admin` database provides roles that are *unavailable* in other databases, including a role that effectively makes a user a MongoDB system superuser. See [Database Administration Roles](#) (page 264) and [Administrative Roles](#) (page 265).

To assign roles to users, you must be a user with administrative role in the database. As such, you must first create an administrative user. For details, see [Create a User Administrator](#) (page 254) and [Add a User to a Database](#) (page 255).

`system.users` Collection

A database's `system.users` (page 268) collection stores information for authentication and authorization to that database. Specifically, the collection stores user credentials for authentication and user privilege information for authorization. MongoDB requires authorization to access the `system.users` (page 268) collection in order to prevent privilege escalation attacks. To access the collection, you must have either `userAdmin` (page 265) or `userAdminAnyDatabase` (page 267) role.

Changed in version 2.4: The schema of `system.users` (page 268) changed to accommodate a more sophisticated authorization using user privilege model, as defined in [privilege documents](#) (page 263).

5.2.2 Inter-Process Authentication

In most cases, *replica set* and *sharded cluster* administrators do not have to keep additional considerations in mind beyond the normal security precautions that all MongoDB administrators must take. However, ensure that:

- Your network configuration will allow every member of the replica set to contact every other member of the replica set.
- If you use MongoDB's authentication system to limit access to your infrastructure, ensure that you configure a `keyFile` on all members to permit authentication.

For most instances, the most effective ways to control access and to secure the connection between members of a *replica set* depend on network-level access control. Use your environment's firewall and network routing to ensure that traffic *only* from clients and other replica set members can reach your `mongod` instances. If needed, use virtual private networks (VPNs) to ensure secure connections over wide area networks (WANs.)

Enable Authentication in Replica Sets and Sharded Clusters

New in version 1.8: Added support authentication in replica set deployments.

Changed in version 1.9.1: Added support authentication in sharded replica set deployments.

MongoDB provides an authentication mechanism for `mongod` and `mongos` instances connecting to replica sets. These instances enable authentication but specify a shared key file that serves as a shared password.

To enable authentication, add the following option to your configuration file:

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

Note: You may choose to set these run-time configuration options using the `--keyFile` (or `mongos --keyFile`) options on the command line.

Setting `keyFile` enables authentication and specifies a key file for the replica set members to use 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 on all `mongos` instances that connect to the set.

The key file must be between 6 and 1024 characters and may only contain characters in the base64 set. The key file must not have group or “world” permissions on UNIX systems. See [Generate a Key File](#) (page 256) for instructions on generating a key file.

5.2.3 Sharded Cluster Security

In most respects security for sharded clusters similar to other MongoDB deployments. Sharded clusters use the same [keyfile](#) (page 236) and [access control](#) (page 235) as all MongoDB deployments. However, there are additional considerations when using authentication with sharded clusters.

Important: In addition to the mechanisms described in this section, always run sharded clusters in a trusted networking environment. Ensure that the network only permits trusted traffic to reach `mongos` and `mongod` instances.

See also:

[Enable Authentication in a Sharded Cluster](#) (page 521).

Access Control Privileges in Sharded Clusters

In sharded clusters, MongoDB provides separate administrative privileges for the sharded cluster and for each shard.

Sharded Cluster Authentication. When connected to a `mongos`, you can grant access to the sharded cluster’s `admin` database.⁶ These credentials reside on the config servers.

Users can access to the cluster according to their [permissions](#) (page 263). To receive privileges for the cluster, you must authenticate while connected to a `mongos` instance.

Shard Server Authentication. To allow administrators to connect and authenticate directly to specific shards, create users in the `admin` database on the `mongod` instance, or *replica set*, that provide each shard.

These users only have access to *a single shard* and are *completely* distinct from the cluster-wide credentials.

Important: Always connect and authenticate to sharded clusters via a `mongos` instance.

Beyond these proprieties, privileges for sharded clusters are functionally the same as any other MongoDB deployment. See [Access Control](#) (page 235) for more information.

Access a Sharded Cluster with Authentication

To access a sharded cluster as an authenticated user, from the command line, use the authentication options when connecting to a `mongos`. Or, you can connect first and then authenticate with the `authenticate` command or the `db.auth()` method.

To close an authenticated session, see the `logout` command.

⁶ Credentials for databases *other* than the `admin` database reside in the `mongod` instance (or sharded cluster) that is the [primary shard](#) (page 495) for that database.

Restriction on localhost Interface

Sharded clusters have restrictions on the use of `localhost` interface. If the host identifier for a MongoDB instance is either `localhost` or “`127.0.0.1`”, then you must use “`localhost`” or “`127.0.0.1`” to identify *all* MongoDB instances in a deployment. This applies to the `host` argument to the `addShard` command as well as to the `--configdb` option for the `mongos`. If you mix `localhost` addresses with remote host address, sharded clusters will not function correctly.

5.2.4 Network Exposure and Security

By default, MongoDB programs (i.e. `mongos` and `mongod`) will bind to all available network interfaces (i.e. IP addresses) on a system.

This page outlines various runtime options that allow you to limit access to MongoDB programs.

Configuration Options

You can limit the network exposure with the following `mongod` and `mongos` configuration options: `nohttpinterface`, `rest`, `bind_ip`, and `port`. You can use a configuration file to specify these settings.

`nohttpinterface`

The `nohttpinterface` setting for `mongod` and `mongos` instances disables the “home” status page, which would run on port 28017 by default. The status interface is read-only by default. You may also specify this option on the command line as `mongod --nohttpinterface` or `mongos --nohttpinterface`.

Authentication does not control or affect access to this interface.

Important: Disable this option for production deployments. If you *do* leave this interface enabled, you should only allow trusted clients to access this port. See [Firewalls](#) (page 239).

`rest`

The `rest` setting for `mongod` enables a fully interactive administrative *REST* interface, which is *disabled by default*. The status interface, which *is* enabled by default, is read-only. This configuration makes that interface fully interactive. The REST interface does not support any authentication and you should always restrict access to this interface to only allow trusted clients to connect to this port.

You may also enable this interface on the command line as `mongod --rest`.

Important: Disable this option for production deployments. If *do* you leave this interface enabled, you should only allow trusted clients to access this port.

`bind_ip`

The `bind_ip` setting for `mongod` and `mongos` instances limits the network interfaces on which MongoDB programs will listen for incoming connections. You can also specify a number of interfaces by passing `bind_ip` a comma separated list of IP addresses. You can use the `mongod --bind_ip` and `mongos --bind_ip` option on the command line at run time to limit the network accessibility of a MongoDB program.

Important: Make sure that your mongod and mongos instances are only accessible on trusted networks. If your system has more than one network interface, bind MongoDB programs to the private or internal network interface.

port

The port setting for mongod and mongos instances changes the main port on which the mongod or mongos instance listens for connections. The default port is 27017. Changing the port does not meaningfully reduce risk or limit exposure. You may also specify this option on the command line as `mongod --port` or `mongos --port`. Setting port also indirectly sets the port for the HTTP status interface, which is always available on the port numbered 1000 greater than the primary mongod port.

Only allow trusted clients to connect to the port for the mongod and mongos instances. See [Firewalls](#) (page 239).

See also [Security Considerations](#) (page 145) and [Default MongoDB Port](#) (page 270).

Firewalls

Firewalls allow administrators to filter and control access to a system by providing granular control over what network communications. For administrators of MongoDB, the following capabilities are important: limiting incoming traffic on a specific port to specific systems, and limiting incoming traffic from untrusted hosts.

On Linux systems, the `iptables` interface provides access to the underlying `netfilter` firewall. On Windows systems, `netsh` command line interface provides access to the underlying Windows Firewall. For additional information about firewall configuration, see [Configure Linux iptables Firewall for MongoDB](#) (page 241) and [Configure Windows netsh Firewall for MongoDB](#) (page 245).

For best results and to minimize overall exposure, ensure that *only* traffic from trusted sources can reach mongod and mongos instances and that the mongod and mongos instances can only connect to trusted outputs.

See also:

For MongoDB deployments on Amazon’s web services, see the [Amazon EC2⁷](#) page, which addresses Amazon’s Security Groups and other EC2-specific security features.

Virtual Private Networks

Virtual private networks, or VPNs, make it possible to link two networks over an encrypted and limited-access trusted network. Typically MongoDB users who use VPNs use SSL rather than IPSEC VPNs for performance issues.

Depending on configuration and implementation, VPNs provide for certificate validation and a choice of encryption protocols, which requires a rigorous level of authentication and identification of all clients. Furthermore, because VPNs provide a secure tunnel, by using a VPN connection to control access to your MongoDB instance, you can prevent tampering and “man-in-the-middle” attacks.

5.2.5 Security and MongoDB API Interfaces

The following section contains strategies to limit risks related to MongoDB’s available interfaces including JavaScript, HTTP, and REST interfaces.

⁷<http://docs.mongodb.org/ecosystem/platforms/amazon-ec2>

JavaScript and the Security of the mongo Shell

The following JavaScript evaluation behaviors of the `mongo` shell represents risk exposures.

JavaScript Expression or JavaScript File

The `mongo` program can evaluate JavaScript expressions using the command line `--eval` option. Also, the `mongo` program can evaluate a JavaScript file (`.js`) passed directly to it (e.g. `mongo someFile.js`).

Because the `mongo` program evaluates the JavaScript without validating the input, this behavior presents a vulnerability.

`.mongorc.js` File

If a `.mongorc.js` file exists⁸, the `mongo` shell will evaluate a `.mongorc.js` file before starting. You can disable this behavior by passing the `mongo --norc` option.

HTTP Status Interface

The HTTP status interface provides a web-based interface that includes a variety of operational data, logs, and status reports regarding the `mongod` or `mongos` instance. The HTTP interface is always available on the port numbered 1000 greater than the primary `mongod` port. By default, the HTTP interface port is 28017, but is indirectly set using the `port` option which allows you to configure the primary `mongod` port.

Without the `rest` setting, this interface is entirely read-only, and limited in scope; nevertheless, this interface may represent an exposure. To disable the HTTP interface, set the `nohttpinterface` run time option or the `--nohttpinterface` command line option. See also [Configuration Options](#) (page 238).

REST API

The REST API to MongoDB provides additional information and write access on top of the HTTP Status interface. While the REST API does not provide any support for insert, update, or remove operations, it does provide administrative access, and its accessibility represents a vulnerability in a secure environment. The REST interface is *disabled* by default, and is not recommended for production use.

If you must use the REST API, please control and limit access to the REST API. The REST API does not include any support for authentication, even when running with `auth` enabled.

See the following documents for instructions on restricting access to the REST API interface:

- [Configure Linux iptables Firewall for MongoDB](#) (page 241)
- [Configure Windows netsh Firewall for MongoDB](#) (page 245)

5.3 Security Tutorials

The following tutorials provide instructions for enabling and using the security features available in MongoDB.

[Network Security Tutorials](#) (page 241) Ensure that the underlying network configuration supports a secure operating environment for MongoDB deployments, and appropriately limits access to MongoDB deployments.

⁸ On Linux and Unix systems, `mongo` reads the `.mongorc.js` file from `$HOME/.mongorc.js` (i.e. `~/mongorc.js`). On Windows, `mongo.exe` reads the `.mongorc.js` file from `%HOMEPATH%\.mongorc.js` or `%HOMEPATH%\mongorc.js`.

[Configure Linux iptables Firewall for MongoDB \(page 241\)](#) Basic firewall configuration patterns and examples for `iptables` on Linux systems.

[Configure Windows netsh Firewall for MongoDB \(page 245\)](#) Basic firewall configuration patterns and examples for `netsh` on Windows systems.

[Connect to MongoDB with SSL \(page 248\)](#) SSL allows MongoDB clients to support encrypted connections to `mongod` instances.

[Access Control Tutorials \(page 253\)](#) MongoDB's access control system provides role-based access control for limiting access to MongoDB deployments. These tutorials describe procedures relevant for the operation and maintenance of this access control system.

[Enable Authentication \(page 253\)](#) Describes the process for enabling authentication for MongoDB deployments.

[Create a User Administrator \(page 254\)](#) Create users with special permissions to create, modify, and remove other users, as well as administer authentication credentials (e.g. passwords).

[Add a User to a Database \(page 255\)](#) Create non-administrator users using MongoDB's role-based authentication system.

[Deploy MongoDB with Kerberos Authentication \(page 257\)](#) Describes the process, for MongoDB Enterprise, used to enable and implement a Kerberos-based authentication system for MongoDB deployments.

[Create a Vulnerability Report \(page 261\)](#) Report a vulnerability in MongoDB.

5.3.1 Network Security Tutorials

The following tutorials provide information on handling network security for MongoDB.

[Configure Linux iptables Firewall for MongoDB \(page 241\)](#) Basic firewall configuration patterns and examples for `iptables` on Linux systems.

[Configure Windows netsh Firewall for MongoDB \(page 245\)](#) Basic firewall configuration patterns and examples for `netsh` on Windows systems.

[Connect to MongoDB with SSL \(page 248\)](#) SSL allows MongoDB clients to support encrypted connections to `mongod` instances.

Configure Linux iptables Firewall for MongoDB

On contemporary Linux systems, the `iptables` program provides methods for managing the Linux Kernel's netfilter or network packet filtering capabilities. These firewall rules make it possible for administrators to control what hosts can connect to the system, and limit risk exposure by limiting the hosts that can connect to a system.

This document outlines basic firewall configurations for `iptables` firewalls on Linux. Use these approaches as a starting point for your larger networking organization. For a detailed overview of security practices and risk management for MongoDB, see [Security Concepts \(page 235\)](#).

See also:

For MongoDB deployments on Amazon's web services, see the [Amazon EC2⁹](#) page, which addresses Amazon's Security Groups and other EC2-specific security features.

⁹<http://docs.mongodb.org/ecosystem/platforms/amazon-ec2>

Overview

Rules in `iptables` configurations fall into chains, which describe the process for filtering and processing specific streams of traffic. Chains have an order, and packets must pass through earlier rules in a chain to reach later rules. This document addresses only the following two chains:

INPUT Controls all incoming traffic.

OUTPUT Controls all outgoing traffic.

Given the [default ports](#) (page 238) of all MongoDB processes, you must configure networking rules that permit *only* required communication between your application and the appropriate `mongod` and `mongos` instances.

Be aware that, by default, the default policy of `iptables` is to allow all connections and traffic unless explicitly disabled. The configuration changes outlined in this document will create rules that explicitly allow traffic from specific addresses and on specific ports, using a default policy that drops all traffic that is not explicitly allowed. When you have properly configured your `iptables` rules to allow only the traffic that you want to permit, you can [Change Default Policy to DROP](#) (page 244).

Patterns

This section contains a number of patterns and examples for configuring `iptables` for use with MongoDB deployments. If you have configured different ports using the `port` configuration setting, you will need to modify the rules accordingly.

Traffic to and from mongod Instances This pattern is applicable to all `mongod` instances running as standalone instances or as part of a *replica set*.

The goal of this pattern is to explicitly allow traffic to the `mongod` instance from the application server. In the following examples, replace `<ip-address>` with the IP address of the application server:

```
iptables -A INPUT -s <ip-address> -p tcp --destination-port 27017 -m state --state NEW,ESTABLISHED -j ACCEPT
iptables -A OUTPUT -d <ip-address> -p tcp --source-port 27017 -m state --state ESTABLISHED -j ACCEPT
```

The first rule allows all incoming traffic from `<ip-address>` on port 27017, which allows the application server to connect to the `mongod` instance. The second rule, allows outgoing traffic from the `mongod` to reach the application server.

Optional

If you have only one application server, you can replace `<ip-address>` with either the IP address itself, such as: 198.51.100.55. You can also express this using CIDR notation as 198.51.100.55/32. If you want to permit a larger block of possible IP addresses you can allow traffic from a <http://docs.mongodb.org/manual/2.4/> using one of the following specifications for the `<ip-address>`, as follows:

```
10.10.10.10/24
10.10.10.10/255.255.255.0
```

Traffic to and from mongos Instances `mongos` instances provide query routing for *sharded clusters*. Clients connect to `mongos` instances, which behave from the client's perspective as `mongod` instances. In turn, the `mongos` connects to all `mongod` instances that are components of the sharded cluster.

Use the same `iptables` command to allow traffic to and from these instances as you would from the `mongod` instances that are members of the replica set. Take the configuration outlined in the [Traffic to and from mongod Instances](#) (page 242) section as an example.

Traffic to and from a MongoDB Config Server Config servers, host the *config database* that stores metadata for sharded clusters. Each production cluster has three config servers, initiated using the `mongod --configsvr` option.¹⁰ Config servers listen for connections on port 27019. As a result, add the following iptables rules to the config server to allow incoming and outgoing connection on port 27019, for connection to the other config servers.

```
iptables -A INPUT -s <ip-address> -p tcp --destination-port 27019 -m state --state NEW,ESTABLISHED -j ACCEPT
iptables -A OUTPUT -d <ip-address> -p tcp --source-port 27019 -m state --state ESTABLISHED -j ACCEPT
```

Replace `<ip-address>` with the address or address space of *all* the mongod that provide config servers.

Additionally, config servers need to allow incoming connections from all of the mongos instances in the cluster *and* all mongod instances in the cluster. Add rules that resemble the following:

```
iptables -A INPUT -s <ip-address> -p tcp --destination-port 27019 -m state --state NEW,ESTABLISHED -j ACCEPT
iptables -A OUTPUT -d <ip-address> -p tcp --source-port 27019 -m state --state ESTABLISHED -j ACCEPT
```

Replace `<ip-address>` with the address of the mongos instances and the shard mongod instances.

Traffic to and from a MongoDB Shard Server For shard servers, running as `mongod --shardsvr`¹¹ Because the default port number when running with shardsvr is 27018, you must configure the following iptables rules to allow traffic to and from each shard:

```
iptables -A INPUT -s <ip-address> -p tcp --destination-port 27018 -m state --state NEW,ESTABLISHED -j ACCEPT
iptables -A OUTPUT -d <ip-address> -p tcp --source-port 27018 -m state --state ESTABLISHED -j ACCEPT
```

Replace the `<ip-address>` specification with the IP address of all mongod. This allows you to permit incoming and outgoing traffic between all shards including constituent replica set members, to:

- all mongod instances in the shard's replica sets.
- all mongod instances in other shards.¹²

Furthermore, shards need to be able make outgoing connections to:

- all mongos instances.
- all mongod instances in the config servers.

Create a rule that resembles the following, and replace the `<ip-address>` with the address of the config servers and the mongos instances:

```
iptables -A OUTPUT -d <ip-address> -p tcp --source-port 27018 -m state --state ESTABLISHED -j ACCEPT
```

Provide Access For Monitoring Systems

1. The mongostat diagnostic tool, when running with the `--discover` needs to be able to reach all components of a cluster, including the config servers, the shard servers, and the mongos instances.
2. If your monitoring system needs access the HTTP interface, insert the following rule to the chain:

```
iptables -A INPUT -s <ip-address> -p tcp --destination-port 28017 -m state --state NEW,ESTABLISHED -j ACCEPT
```

Replace `<ip-address>` with the address of the instance that needs access to the HTTP or REST interface. For *all* deployments, you should restrict access to this port to *only* the monitoring instance.

Optional

For shard server mongod instances running with shardsvr, the rule would resemble the following:

¹⁰ You can also run a config server by setting the `configsvr` option in a configuration file.

¹¹ You can also specify the shard server option using the `shardsvr` setting in the configuration file. Shard members are also often conventional replica sets using the default port.

¹² All shards in a cluster need to be able to communicate with all other shards to facilitate *chunk* and balancing operations.

```
iptables -A INPUT -s <ip-address> -p tcp --destination-port 28018 -m state --state NEW,ESTABLISH
```

For config server mongod instances running with configsvr, the rule would resemble the following:

```
iptables -A INPUT -s <ip-address> -p tcp --destination-port 28019 -m state --state NEW,ESTABLISH
```

Change Default Policy to DROP

The default policy for `iptables` chains is to allow all traffic. After completing all `iptables` configuration changes, you *must* change the default policy to `DROP` so that all traffic that isn't explicitly allowed as above will not be able to reach components of the MongoDB deployment. Issue the following commands to change this policy:

```
iptables -P INPUT DROP  
iptables -P OUTPUT DROP
```

Manage and Maintain `iptables` Configuration

This section contains a number of basic operations for managing and using `iptables`. There are various front end tools that automate some aspects of `iptables` configuration, but at the core all `iptables` front ends provide the same basic functionality:

Make all `iptables` Rules Persistent By default all `iptables` rules are only stored in memory. When your system restarts, your firewall rules will revert to their defaults. When you have tested a rule set and have guaranteed that it effectively controls traffic you can use the following operations to you should make the rule set persistent.

On Red Hat Enterprise Linux, Fedora Linux, and related distributions you can issue the following command:

```
service iptables save
```

On Debian, Ubuntu, and related distributions, you can use the following command to dump the `iptables` rules to the `/etc/iptables.conf` file:

```
iptables-save > /etc/iptables.conf
```

Run the following operation to restore the network rules:

```
iptables-restore < /etc/iptables.conf
```

Place this command in your `rc.local` file, or in the `/etc/network/if-up.d/iptables` file with other similar operations.

List all `iptables` Rules To list all of currently applied `iptables` rules, use the following operation at the system shell.

```
iptables --L
```

Flush all `iptables` Rules If you make a configuration mistake when entering `iptables` rules or simply need to revert to the default rule set, you can use the following operation at the system shell to flush all rules:

```
iptables --F
```

If you've already made your `iptables` rules persistent, you will need to repeat the appropriate procedure in the [Make all iptables Rules Persistent](#) (page 244) section.

Configure Windows `netsh` Firewall for MongoDB

On Windows Server systems, the `netsh` program provides methods for managing the *Windows Firewall*. These firewall rules make it possible for administrators to control what hosts can connect to the system, and limit risk exposure by limiting the hosts that can connect to a system.

This document outlines basic *Windows Firewall* configurations. Use these approaches as a starting point for your larger networking organization. For a detailed overview of security practices and risk management for MongoDB, see [Security Concepts](#) (page 235).

See also:

[Windows Firewall](#)¹³ documentation from Microsoft.

Overview

Windows Firewall processes rules in an ordered determined by rule type, and parsed in the following order:

1. Windows Service Hardening
2. Connection security rules
3. Authenticated Bypass Rules
4. Block Rules
5. Allow Rules
6. Default Rules

By default, the policy in *Windows Firewall* allows all outbound connections and blocks all incoming connections.

Given the [default ports](#) (page 238) of all MongoDB processes, you must configure networking rules that permit *only* required communication between your application and the appropriate `mongod.exe` and `mongos.exe` instances.

The configuration changes outlined in this document will create rules which explicitly allow traffic from specific addresses and on specific ports, using a default policy that drops all traffic that is not explicitly allowed.

You can configure the *Windows Firewall* with using the `netsh` command line tool or through a windows application. On Windows Server 2008 this application is *Windows Firewall With Advanced Security* in *Administrative Tools*. On previous versions of Windows Server, access the *Windows Firewall* application in the *System and Security* control panel.

The procedures in this document use the `netsh` command line tool.

Patterns

This section contains a number of patterns and examples for configuring *Windows Firewall* for use with MongoDB deployments. If you have configured different ports using the `port` configuration setting, you will need to modify the rules accordingly.

¹³<http://technet.microsoft.com/en-us/network/bb545423.aspx>

Traffic to and from mongod.exe Instances This pattern is applicable to all mongod.exe instances running as standalone instances or as part of a *replica set*. The goal of this pattern is to explicitly allow traffic to the mongod.exe instance from the application server.

```
netsh advfirewall firewall add rule name="Open mongod port 27017" dir=in action=allow protocol=TCP loca
```

This rule allows all incoming traffic to port 27017, which allows the application server to connect to the mongod.exe instance.

Windows Firewall also allows enabling network access for an entire application rather than to a specific port, as in the following example:

```
netsh advfirewall firewall add rule name="Allowing mongod" dir=in action=allow program=" C:\mongodb\bin\mongod.exe"
```

You can allow all access for a mongos.exe server, with the following invocation:

```
netsh advfirewall firewall add rule name="Allowing mongos" dir=in action=allow program=" C:\mongodb\bin\mongos.exe"
```

Traffic to and from mongos.exe Instances mongos.exe instances provide query routing for *sharded clusters*. Clients connect to mongos.exe instances, which behave from the client's perspective as mongod.exe instances. In turn, the mongos.exe connects to all mongod.exe instances that are components of the sharded cluster.

Use the same *Windows Firewall* command to allow traffic to and from these instances as you would from the mongod.exe instances that are members of the replica set.

```
netsh advfirewall firewall add rule name="Open mongod shard port 27018" dir=in action=allow protocol=TCP loca
```

Traffic to and from a MongoDB Config Server Configuration servers, host the *config database* that stores metadata for sharded clusters. Each production cluster has three configuration servers, initiated using the *mongod --configsvr* option.¹⁴ Configuration servers listen for connections on port 27019. As a result, add the following *Windows Firewall* rules to the config server to allow incoming and outgoing connection on port 27019, for connection to the other config servers.

```
netsh advfirewall firewall add rule name="Open mongod config svr port 27019" dir=in action=allow protocol=TCP loca
```

Additionally, config servers need to allow incoming connections from all of the mongos.exe instances in the cluster and all mongod.exe instances in the cluster. Add rules that resemble the following:

```
netsh advfirewall firewall add rule name="Open mongod config svr inbound" dir=in action=allow protocol=TCP loca
```

Replace <ip-address> with the addresses of the mongos.exe instances and the shard mongod.exe instances.

Traffic to and from a MongoDB Shard Server For shard servers, running as *mongod --shardsvr*¹⁵ Because the default port number when running with shardsvr is 27018, you must configure the following *Windows Firewall* rules to allow traffic to and from each shard:

```
netsh advfirewall firewall add rule name="Open mongod shardsvr inbound" dir=in action=allow protocol=TCP loca
netsh advfirewall firewall add rule name="Open mongod shardsvr outbound" dir=out action=allow protocol=TCP loca
```

Replace the <ip-address> specification with the IP address of all mongod.exe instances. This allows you to permit incoming and outgoing traffic between all shards including constituent replica set members to:

- all mongod.exe instances in the shard's replica sets.

¹⁴ You can also run a config server by setting the *configsvr* option in a configuration file.

¹⁵ You can also specify the shard server option using the *shardsvr* setting in the configuration file. Shard members are also often conventional replica sets using the default port.

- all mongod.exe instances in other shards.¹⁶

Furthermore, shards need to be able make outgoing connections to:

- all mongos.exe instances.
- all mongod.exe instances in the config servers.

Create a rule that resembles the following, and replace the <ip-address> with the address of the config servers and the mongos.exe instances:

```
netsh advfirewall firewall add rule name="Open mongod config svr outbound" dir=out action=allow proto=tcp localport=27017 remoteip=<ip-address>
```

Provide Access For Monitoring Systems

1. The mongostat diagnostic tool, when running with the --discover needs to be able to reach all components of a cluster, including the config servers, the shard servers, and the mongos.exe instances.
2. If your monitoring system needs access the HTTP interface, insert the following rule to the chain:

```
netsh advfirewall firewall add rule name="Open mongod HTTP monitoring inbound" dir=in action=allow proto=tcp localport=28012
```

Replace <ip-address> with the address of the instance that needs access to the HTTP or REST interface. For *all* deployments, you should restrict access to this port to *only* the monitoring instance.

Optional

For shard server mongod.exe instances running with shardsvr, the rule would resemble the following:

```
netsh advfirewall firewall add rule name="Open mongos HTTP monitoring inbound" dir=in action=allow proto=tcp localport=28012
```

For config server mongod.exe instances running with configsrv, the rule would resemble the following:

```
netsh advfirewall firewall add rule name="Open mongod configsvr HTTP monitoring inbound" dir=in action=allow proto=tcp localport=28012
```

Manage and Maintain Windows Firewall Configurations

This section contains a number of basic operations for managing and using netsh. While you can use the GUI front ends to manage the *Windows Firewall*, all core functionality is accessible from netsh.

Delete all Windows Firewall Rules To delete the firewall rule allowing mongod.exe traffic:

```
netsh advfirewall firewall delete rule name="Open mongod port 27017" protocol=tcp localport=27017
```

```
netsh advfirewall firewall delete rule name="Open mongod shard port 27018" protocol=tcp localport=27018
```

List All Windows Firewall Rules To return a list of all Windows Firewall rules:

```
netsh advfirewall firewall show rule name=all
```

Reset Windows Firewall To reset the Windows Firewall rules:

```
netsh advfirewall reset
```

¹⁶ All shards in a cluster need to be able to communicate with all other shards to facilitate *chunk* and balancing operations.

Backup and Restore Windows Firewall Rules To simplify administration of larger collection of systems, you can export or import firewall systems from different servers) rules very easily on Windows:

Export all firewall rules with the following command:

```
netsh advfirewall export "C:\temp\MongoDBfw.wfw"
```

Replace "C:\temp\MongoDBfw.wfw" with a path of your choosing. You can use a command in the following form to import a file created using this operation:

```
netsh advfirewall import "C:\temp\MongoDBfw.wfw"
```

Connect to MongoDB with SSL

This document outlines the use and operation of MongoDB's SSL support. SSL allows MongoDB clients to support encrypted connections to mongod instances.

Note: The default distribution of MongoDB¹⁷ does **not** contain support for SSL. To use SSL, you must either build MongoDB locally passing the “--ssl” option to scons or use MongoDB Enterprise¹⁸.

These instructions outline the process for getting started with SSL and assume that you have already installed a build of MongoDB that includes SSL support and that your client driver supports SSL.

Configure mongod and mongos for SSL

Combine SSL Certificate and Key File Before you can use SSL, you must have a .pem file that contains the public key certificate and private key. MongoDB can use any valid SSL certificate. To generate a self-signed certificate and private key, use a command that resembles the following:

```
cd /etc/ssl/
openssl req -new -x509 -days 365 -nodes -out mongodb-cert.crt -keyout mongodb-cert.key
```

This operation generates a new, self-signed certificate with no passphrase that is valid for 365 days. Once you have the certificate, concatenate the certificate and private key to a .pem file, as in the following example:

```
cat mongodb-cert.key mongodb-cert.crt > mongodb.pem
```

Set Up mongod and mongos with SSL Certificate and Key To use SSL in your MongoDB deployment, include the following run-time options with mongod and mongos:

- sslOnNormalPorts
- sslPEMKeyFile with the .pem file that contains the SSL certificate and key.

Consider the following syntax for mongod:

```
mongod --sslOnNormalPorts --sslPEMKeyFile <pem>
```

For example, given an SSL certificate located at /etc/ssl/mongodb.pem, configure mongod to use SSL encryption for all connections with the following command:

```
mongod --sslOnNormalPorts --sslPEMKeyFile /etc/ssl/mongodb.pem
```

Note:

¹⁷<http://www.mongodb.org/downloads>

¹⁸<http://www.mongodb.com/products/mongodb-enterprise>

- Specify <pem> with the full path name to the certificate.
- If the private key portion of the <pem> is encrypted, specify the encryption password with the `sslPEMKeyPassword` option.
- You may also specify these options in the configuration file, as in the following example:

```
sslOnNormalPorts = true
sslPEMKeyFile = /etc/ssl/mongodb.pem
```

To connect, to mongod and mongos instances using SSL, the mongo shell and MongoDB tools must include the `--ssl` option. See [SSL Configuration for Clients](#) (page 250) for more information on connecting to mongod and mongos running with SSL.

Set Up mongod and mongos with Certificate Validation To set up mongod or mongos for SSL encryption using an SSL certificate signed by a certificate authority, include the following run-time options during startup:

- `sslOnNormalPorts`
- `sslPEMKeyFile` with the name of the .pem file that contains the signed SSL certificate and key.
- `sslCAFile` with the name of the .pem file that contains the root certificate chain from the Certificate Authority.

Consider the following syntax for mongod:

```
mongod --sslOnNormalPorts --sslPEMKeyFile <pem> --sslCAFile <ca>
```

For example, given a signed SSL certificate located at `/etc/ssl/mongodb.pem` and the certificate authority file at `/etc/ssl/ca.pem`, you can configure mongod for SSL encryption as follows:

```
mongod --sslOnNormalPorts --sslPEMKeyFile /etc/ssl/mongodb.pem --sslCAFile /etc/ssl/ca.pem
```

Note:

- Specify the <pem> file and the <ca> file with either the full path name or the relative path name.
- If the <pem> is encrypted, specify the encryption password with the `sslPEMKeyPassword` option.
- You may also specify these options in the configuration file, as in the following example:

```
sslOnNormalPorts = true
sslPEMKeyFile = /etc/ssl/mongodb.pem
sslCAFile = /etc/ssl/ca.pem
```

To connect, to mongod and mongos instances using SSL, the mongo tools must include the both the `--ssl` and `--sslPEMKeyFile` option. See [SSL Configuration for Clients](#) (page 250) for more information on connecting to mongod and mongos running with SSL.

Block Revoked Certificates for Clients To prevent clients with revoked certificates from connecting, include the `sslCRLFile` to specify a .pem file that contains revoked certificates.

For example, the following mongod with SSL configuration includes the `sslCRLFile` setting:

```
mongod --sslOnNormalPorts --sslCRLFile /etc/ssl/ca-crl.pem --sslPEMKeyFile /etc/ssl/mongodb.pem --ss...
```

Clients with revoked certificates in the `/etc/ssl/ca-crl.pem` will not be able to connect to this mongod instance.

Validate Only if a Client Presents a Certificate In most cases it is important to ensure that clients present valid certificates. However, if you have clients that cannot present a client certificate, or are transitioning to using a certificate authority you may only want to validate certificates from clients that present a certificate.

If you want to bypass validation for clients that don't present certificates, include the `sslWeakCertificateValidation` run-time option with `mongod` and `mongos`. If the client does not present a certificate, no validation occurs. These connections, though not validated, are still encrypted using SSL.

For example, consider the following `mongod` with an SSL configuration that includes the `sslWeakCertificateValidation` setting:

```
mongod --sslOnNormalPorts --sslWeakCertificateValidation --sslPEMKeyFile /etc/ssl/mongodb.pem --sslCAFile /etc/ssl/ca.pem
```

Then, clients can connect either with the option `--ssl` and **no** certificate or with the option `--ssl` and a **valid** certificate. See [SSL Configuration for Clients](#) (page 250) for more information on SSL connections for clients.

Note: If the client presents a certificate, the certificate must be a valid certificate.

All connections, including those that have not presented certificates are encrypted using SSL.

Run in FIPS Mode If your `mongod` or `mongos` is running on a system with an OpenSSL library configured with the FIPS 140-2 module, you can run `mongod` or `mongos` in FIPS mode, with the `sslFIPSMode` setting.

SSL Configuration for Clients

Clients must have support for SSL to work with a `mongod` or a `mongos` instance that has SSL support enabled. The current versions of the Python, Java, Ruby, Node.js, .NET, and C++ drivers have support for SSL, with full support coming in future releases of other drivers.

mongo SSL Configuration For SSL connections, you must use the `mongo` shell built with SSL support or distributed with MongoDB Enterprise. To support SSL, `mongo` has the following settings:

- `--ssl`
- `--sslPEMKeyFile` with the name of the `.pem` file that contains the SSL certificate and key.
- `--sslCAFile` with the name of the `.pem` file that contains the certificate from the Certificate Authority.
- `--sslPEMKeyPassword` option if the client certificate-key file is encrypted.

Connect to MongoDB Instance with SSL Encryption To connect to a `mongod` or `mongos` instance that requires *only a SSL encryption mode* (page 248), start `mongo` shell with `--ssl`, as in the following:

```
mongo --ssl
```

Connect to MongoDB Instance that Requires Client Certificates To connect to a `mongod` or `mongos` that requires *CA-signed client certificates* (page 249), start the `mongo` shell with `--ssl` and the `--sslPEMKeyFile` option to specify the signed certificate-key file, as in the following:

```
mongo --ssl --sslPEMKeyFile /etc/ssl/client.pem
```

Connect to MongoDB Instance that Validates when Presented with a Certificate To connect to a mongod or mongos instance that *only requires valid certificates when the client presents a certificate* (page 250), start mongo shell either with the `--ssl` ssl and **no** certificate or with the `--ssl` ssl and a **valid** signed certificate.

For example, if mongod is running with weak certificate validation, both of the following mongo shell clients can connect to that mongod:

```
mongo --ssl
mongo --ssl --sslPEMKeyFile /etc/ssl/client.pem
```

Important: If the client presents a certificate, the certificate must be valid.

MMS Monitoring Agent The Monitoring agent will also have to connect via SSL in order to gather its stats. Because the agent already utilizes SSL for its communications to the MMS servers, this is just a matter of enabling SSL support in MMS itself on a per host basis.

Use the “Edit” host button (i.e. the pencil) on the Hosts page in the MMS console to enable SSL.

Please see the [MMS documentation](#)¹⁹ for more information about MMS configuration.

PyMongo Add the “ssl=True” parameter to a PyMongo `MongoClient`²⁰ to create a MongoDB connection to an SSL MongoDB instance:

```
from pymongo import MongoClient
c = MongoClient(host="mongodb.example.net", port=27017, ssl=True)
```

To connect to a replica set, use the following operation:

```
from pymongo import MongoReplicaSetClient
c = MongoReplicaSetClient("mongodb.example.net:27017",
                          replicaSet="mysetName", ssl=True)
```

PyMongo also supports an “ssl=true” option for the MongoDB URI:

```
mongodb://mongodb.example.net:27017/?ssl=true
```

Java Consider the following example “SSLApp.java” class file:

```
import com.mongodb.*;
import javax.net.ssl.SSLSocketFactory;

public class SSLApp {

    public static void main(String args[])
        throws Exception {

        MongoClientOptions o = new MongoClientOptions.Builder()
            .socketFactory(SSLocketFactory.getDefault())
            .build();

        MongoClient m = new MongoClient("localhost", o);

        DB db = m.getDB("test");
        DBCollection c = db.getCollection("foo");
```

¹⁹<http://mms.mongodb.com/help>

²⁰http://api.mongodb.org/python/current/api/pymongo/mongo_client.html#pymongo.mongo_client.MongoClient

```
        System.out.println( c.findOne() );
    }
}
```

Ruby The recent versions of the Ruby driver have support for connections to SSL servers. Install the latest version of the driver with the following command:

```
gem install mongo
```

Then connect to a standalone instance, using the following form:

```
require 'rubygems'
require 'mongo'

connection = MongoClient.new('localhost', 27017, :ssl => true)
```

Replace connection with the following if you're connecting to a replica set:

```
connection = MongoReplicaSetClient.new(['localhost:27017'],
                                         ['localhost:27018'],
                                         :ssl => true)
```

Here, mongod instance run on “localhost:27017” and “localhost:27018”.

Node.JS (node-mongodb-native) In the `node-mongodb-native`²¹ driver, use the following invocation to connect to a mongod or mongos instance via SSL:

```
var db1 = new Db(MONGODB, new Server("127.0.0.1", 27017,
                                      { auto_reconnect: false, poolSize: 4, ssl:ssl } ));
```

To connect to a replica set via SSL, use the following form:

```
var replSet = new ReplSetServers(
  new Server( RS.host, RS.ports[1], { auto_reconnect: true } ),
  new Server( RS.host, RS.ports[0], { auto_reconnect: true } ),
),
{rs_name:RS.name, ssl:ssl}
);
```

.NET As of release 1.6, the .NET driver supports SSL connections with mongod and mongos instances. To connect using SSL, you must add an option to the connection string, specifying `ssl=true` as follows:

```
var connectionString = "mongodb://localhost/?ssl=true";
var server = MongoServer.Create(connectionString);
```

The .NET driver will validate the certificate against the local trusted certificate store, in addition to providing encryption of the server. This behavior may produce issues during testing if the server uses a self-signed certificate. If you encounter this issue, add the `sslverifycertificate=false` option to the connection string to prevent the .NET driver from validating the certificate, as follows:

```
var connectionString = "mongodb://localhost/?ssl=true&sslverifycertificate=false";
var server = MongoServer.Create(connectionString);
```

²¹<https://github.com/mongodb/node-mongodb-native>

5.3.2 Access Control Tutorials

The following tutorials provide instructions on how to enable authentication and limit access for users with privilege roles.

[Enable Authentication \(page 253\)](#) Describes the process for enabling authentication for MongoDB deployments.

[Create a User Administrator \(page 254\)](#) Create users with special permissions to create, modify, and remove other users, as well as administer authentication credentials (e.g. passwords).

[Add a User to a Database \(page 255\)](#) Create non-administrator users using MongoDB's role-based authentication system.

[Change a User's Password \(page 256\)](#) Only user administrators can edit credentials. This tutorial describes the process for editing an existing user's password.

[Generate a Key File \(page 256\)](#) Use key file to allow the components of MongoDB sharded cluster or replica set to mutually authenticate.

[Deploy MongoDB with Kerberos Authentication \(page 257\)](#) Describes the process, for MongoDB Enterprise, used to enable and implement a Kerberos-based authentication system for MongoDB deployments.

Enable Authentication

Enable authentication using the `auth` or `keyFile` settings. Use `auth` for standalone instances, and `keyFile` with *replica sets* and *sharded clusters*. `keyFile` implies `auth` and allows members of a MongoDB deployment to authenticate internally.

Authentication requires at least one administrator user in the `admin` database. You can create the user before enabling authentication or after enabling authentication.

See also:

[Deploy MongoDB with Kerberos Authentication \(page 257\)](#).

Procedures

You can enable authentication using either of the following procedures, depending

Create the Administrator Credentials and then Enable Authentication

1. Start the `mongod` or `mongos` instance *without* the `auth` or `keyFile` setting.
2. Create the administrator user as described in [Create a User Administrator \(page 254\)](#).
3. Re-start the `mongod` or `mongos` instance with the `auth` or `keyFile` setting.

Enable Authentication and then Create Administrator

1. Start the `mongod` or `mongos` instance with the `auth` or `keyFile` setting.
2. Connect to the instance on the same system so that you can authenticate using the [localhost exception \(page 255\)](#).
3. Create the administrator user as described in [Create a User Administrator \(page 254\)](#).

Query Authenticated Users

If you have the [userAdmin](#) (page 265) or [userAdminAnyDatabase](#) (page 267) role on a database, you can query authenticated users in that database with the following operation:

```
db.system.users.find()
```

Create a User Administrator

In a MongoDB deployment, users with either the [userAdmin](#) (page 265) or [userAdminAnyDatabase](#) (page 267) roles are *effective* administrative “superusers”. Users with either of these roles can create and modify any other users and can assign them any privileges. The user also can grant *itself* any privileges. In production deployments, this user should have *no other roles* and should only administer users and privileges.

This should be the first user created for a MongoDB deployment. This user can then create all other users in the system.

Important: The [userAdminAnyDatabase](#) (page 267) user can grant itself and any other user full access to the entire MongoDB instance. The credentials to log in as this user should be carefully controlled.

Users with the [userAdmin](#) (page 265) and [userAdminAnyDatabase](#) (page 267) privileges are not the same as the UNIX `root` superuser in that this role confers **no additional access** beyond user administration. These users cannot perform administrative operations or read or write data without first conferring themselves with additional permissions.

Note: The [userAdmin](#) (page 265) role is a database-specific privilege, and *only* grants a user the ability to administer users on a single database. However, for the `admin` database, [userAdmin](#) (page 265) allows a user the ability to gain [userAdminAnyDatabase](#) (page 267). Thus, for the `admin` database **only**, these roles are effectively the same.

Create a User Administrator

1. Connect to the `mongod` or `mongos` by either:

- Authenticating as an existing user with the [userAdmin](#) (page 265) or [userAdminAnyDatabase](#) (page 267) role.
- Authenticating using the [localhost exception](#) (page 255). When creating the first user in a deployment, you must authenticate using the [localhost exception](#) (page 255).

2. Switch to the `admin` database:

```
db = db.getSiblingDB('admin')
```

3. Add the user with either the [userAdmin](#) (page 265) role or [userAdminAnyDatabase](#) (page 267) role, and only that role, by issuing a command similar to the following, where `<username>` is the username and `<password>` is the password:

```
db.addUser( { user: "<username>",
              pwd: "<password>",
              roles: [ "userAdminAnyDatabase" ] } )
```

To authenticate as this user, you must authenticate against the `admin` database.

Authenticate with Full Administrative Access via Localhost

If there are no users for the `admin` database, you can connect with full administrative access via the localhost interface. This bypass exists to support bootstrapping new deployments. This approach is useful, for example, if you want to run `mongod` or `mongos` with authentication before creating your first user.

To authenticate via localhost, connect to the `mongod` or `mongos` from a client running on the same system. Your connection will have full administrative access.

To disable the localhost bypass, set the `enableLocalhostAuthBypass` parameter using `setParameter` during startup:

```
mongod --setParameter enableLocalhostAuthBypass=0
```

Note: For versions of MongoDB 2.2 prior to 2.2.4, if `mongos` is running with `keyFile`, then all users connecting over the localhost interface must authenticate, even if there aren't any users in the `admin` database. Connections on localhost are not correctly granted full access on sharded systems that run those versions.

MongoDB 2.2.4 resolves this issue.

Note: In version 2.2, you cannot add the first user to a sharded cluster using the `localhost` connection. If you are running a 2.2 sharded cluster and want to enable authentication, you must deploy the cluster and add the first user to the `admin` database before restarting the cluster to run with `keyFile`.

Add a User to a Database

To add a user to a database you must authenticate to that database as a user with the `userAdmin` (page 265) or `userAdminAnyDatabase` (page 267) role. If you have not first created a user with one of those roles, do so as described in [Create a User Administrator](#) (page 254).

When adding a user to multiple databases, you must define the user *for each database*. See [Password Hashing Insecurity](#) (page 271) for important security information.

To add a user, pass the `db.addUser()` method a well formed [privilege document](#) (page 263) that contains the user's credentials and privileges. The `db.addUser()` method adds the document to the database's `system.users` (page 268) collection.

Changed in version 2.4: In previous versions of MongoDB, you could change an existing user's password by calling `db.addUser()` again with the user's username and their updated password. Anything specified in the `addUser()` method would override the existing information for that user. In newer versions of MongoDB, this will result in a duplicate key error.

To change a user's password in version 2.4 or newer, see [Change a User's Password](#) (page 256).

For the structure of a privilege document, see `system.users` (page 268). For descriptions of user roles, see [User Privilege Roles in MongoDB](#) (page 263).

Example

The following creates a user named Alice in the `products` database and gives her `readWrite` and `dbAdmin` privileges.

```
use products
db.addUser( { user: "Alice",
              pwd: "Moon1234",
              roles: [ "readWrite", "dbAdmin" ]
            } )
```

Example

The following creates a user named Bob in the admin database. The [privilege document](#) (page 268) uses Bob's credentials from the products database and assigns him userAdmin privileges.

```
use admin
db.addUser( { user: "Bob",
              userSource: "products",
              roles: [ "userAdmin" ]
            } )
```

Example

The following creates a user named Carlos in the admin database and gives him readWrite access to the config database, which lets him change certain settings for sharded clusters, such as to disable the balancer.

```
db = db.getSiblingDB('admin')
db.addUser( { user: "Carlos",
              pwd: "Moon1234",
              roles: [ "clusterAdmin" ],
              otherDBRoles: { config: [ "readWrite" ] }
            } )
```

Only the admin database supports the [otherDBRoles](#) (page 269) field.

Change a User's Password

New in version 2.4.

To change a user's password, you must have the [userAdmin](#) (page 265) role on the database that contains the definition of the user whose password you wish to change.

To update the password, pass the user's username and the new desired password to the `db.changeUserPassword()` method.

Example

The following operation changes the reporting user's password to SOhSS3TbYhxusooLiW8ypJPxmt1oOfL:

```
db = db.getSiblingDB('records')
db.changeUserPassword("reporting", "SOhSS3TbYhxusooLiW8ypJPxmt1oOfL")
```

Note: In previous versions of MongoDB, you could change an existing user's password by calling `db.addUser()` again with the user's username and their updated password. Anything specified in the `addUser()` method would override the existing information for that user. In newer versions of MongoDB, this will result in a duplicate key error.

For more about changing a user's password prior to version 2.4, see: [Add a User to a Database](#) (page 255).

Generate a Key File

This section describes how to generate a key file to store authentication information. After generating a key file, specify the key file using the `keyFile` option when starting a `mongod` or `mongos` instance.

A key's length must be between 6 and 1024 characters and may only contain characters in the base64 set. The key file must not have group or world permissions on UNIX systems. Key file permissions are not checked on Windows systems.

Generate a Key File

Use the following `openssl` command at the system shell to generate pseudo-random content for a key file:

```
openssl rand -base64 741
```

Note: Key file permissions are not checked on Windows systems.

Key File Properties

Be aware that MongoDB strips whitespace characters (e.g. `x0d`, `x09`, and `x20`,) for cross-platform convenience. As a result, the following operations produce identical keys:

```
echo -e "my secret key" > key1
echo -e "my secret key\n" > key2
echo -e "my      secret      key" > key3
echo -e "my\r\nsecret\r\nkey\r\n" > key4
```

Deploy MongoDB with Kerberos Authentication

New in version 2.4.

MongoDB Enterprise supports authentication using a Kerberos service. Kerberos is an industry standard authentication protocol for large client/server system. With Kerberos MongoDB and application ecosystems can take advantage of existing authentication infrastructure and processes.

Setting up and configuring a Kerberos deployment is beyond the scope of this document. In order to use MongoDB with Kerberos, you must have a properly configured Kerberos deployment and the ability to generate a valid `keytab` file for each `mongod` instance in your MongoDB deployment.

Note: The following assumes that you have a valid Kerberos keytab file for your realm accessible on your system. The examples below assume that the keytab file is valid and is located at `http://docs.mongodb.org/manual/opt/mongodb/mongod.keytab` and is *only* accessible to the user that runs the `mongod` process.

Process Overview

To run MongoDB with Kerberos support, you must:

- Configure a Kerberos service principal for each `mongod` and `mongos` instance in your MongoDB deployment.
- Generate and distribute keytab files for each MongoDB component (i.e. `mongod` and `mongos`) in your deployment. Ensure that you *only* transmit keytab files over secure channels.
- Optional. Start the `mongod` instance *without* `auth` and create users inside of MongoDB that you can use to bootstrap your deployment.
- Start `mongod` and `mongos` with the `KRB5_KTNAME` environment variable as well as a number of required run time options.

- If you did not create Kerberos user accounts, you can use the *localhost exception* (page 255) to create users at this point until you create the first user on the `admin` database.
- Authenticate clients, including the `mongo` shell using Kerberos.

Operations

Create Users and Privilege Documents For every user that you want to be able to authenticate using Kerberos, you must create corresponding privilege documents in the `system.users` (page 268) collection to provision access to users. Consider the following document:

```
{  
    user: "application/reporting@EXAMPLE.NET",  
    roles: ["read"],  
    userSource: "$external"  
}
```

This grants the Kerberos user principal `application/reporting@EXAMPLE.NET` read only access to a database. The `userSource` (page 269) `$external` reference allows `mongod` to consult an external source (i.e. Kerberos) to authenticate this user.

In the `mongo` shell you can pass the `db.addUser()` a user privilege document to provision access to users, as in the following operation:

```
db = db.getSiblingDB("records")  
db.addUser( {  
    "user": "application/reporting@EXAMPLE.NET",  
    "roles": [ "read" ],  
    "userSource": "$external"  
} )
```

These operations grants the Kerberos user `application/reporting@EXAMPLE.NET` access to the `records` database.

To remove access to a user, use the `remove()` method, as in the following example:

```
db.system.users.remove( { user: "application/reporting@EXAMPLE.NET" } )
```

To modify a user document, use `update` (page 40) operations on documents in the `system.users` (page 268) collection.

See also:

`system.users` Privilege Documents (page 268) and `User Privilege Roles in MongoDB` (page 263).

Start `mongod` with Kerberos Support Once you have provisioned privileges to users in the `mongod`, *and* obtained a valid keytab file, you must start `mongod` using a command in the following form:

```
env KRB5_KTNAME=<path to keytab file> <mongod invocation>
```

For successful operation with `mongod` use the following run time options in addition to your normal default configuration options:

- `--setParameter` with the `authenticationMechanisms=GSSAPI` argument to enable support for Kerberos.
- `--auth` to enable authentication.
- `--keyFile` to allow components of a single MongoDB deployment to communicate with each other, if needed to support replica set and sharded cluster operations. `keyFile` implies `auth`.

For example, consider the following invocation:

```
env KRB5_KTNAME=/opt/mongodb/mongod.keytab \
    /opt/mongodb/bin/mongod --dbpath /opt/mongodb/data \
    --fork --logpath /opt/mongodb/log/mongod.log \
    --auth --setParameter authenticationMechanisms=GSSAPI
```

You can also specify these options using the configuration file. As in the following:

```
# /opt/mongodb/mongod.conf, Example configuration file.

fork = true
auth = true

dbpath = /opt/mongodb/data
logpath = /opt/mongodb/log/mongod.log
setParameter = authenticationMechanisms=GSSAPI
```

To use this configuration file, start mongod as in the following:

```
env KRB5_KTNAME=/opt/mongodb/mongod.keytab \
    /opt/mongodb/bin/mongod --config /opt/mongodb/mongod.conf
```

To start a mongos instance using Kerberos, you must create a Kerberos service principal and deploy a keytab file for this instance, and then start the mongos with the following invocation:

```
env KRB5_KTNAME=/opt/mongodb/mongos.keytab \
    /opt/mongodb/bin/mongos
    --configdb shard0.example.net,shard1.example.net,shard2.example.net \
    --setParameter authenticationMechanisms=GSSAPI \
    --keyFile /opt/mongodb/mongos.keyfile
```

Tip

If you installed MongoDB Enterprise using one of the official .deb or .rpm packages and are controlling the mongod instance using the included init/upstart scripts, you can set the KRB5_KTNAME variable in the default environment settings file. For .rpm packages this file is located at /etc/sysconfig/mongod. For .deb packages, this file is /etc/default/mongodb. Set the value in a line that resembles the following:

```
export KRB5_KTNAME=<setting>"
```

If you encounter problems when trying to start mongod or mongos, please see the [troubleshooting section](#) (page 260) for more information.

Important: Before users can authenticate to MongoDB using Kerberos you must [create users](#) (page 258) and grant them privileges within MongoDB. If you have not created users when you start MongoDB with Kerberos you can use the [localhost authentication exception](#) (page 255) to add users. See the [Create Users and Privilege Documents](#) (page 258) section and the [User Privilege Roles in MongoDB](#) (page 263) document for more information.

Authenticate mongo Shell with Kerberos To connect to a mongod instance using the mongo shell you must begin by using the kinit program to initialize and authenticate a Kerberos session. Then, start a mongo instance, and use the db.auth() method, to authenticate against the special \$external database, as in the following operation:

```
use $external
db.auth( { mechanism: "GSSAPI", user: "application/reporting@EXAMPLE.NET" } )
```

Alternately, you can authenticate using command line options to mongo, as in the following equivalent example:

```
mongo --authenticationMechanism=GSSAPI
  --authenticationDatabase='$external' \
  --username application/reporting@EXAMPLE.NET
```

These operations authenticate the Kerberos principal name `application/reporting@EXAMPLE.NET` to the connected `mongod`, and will automatically acquire all available privileges as needed.

Use MongoDB Drivers to Authenticate with Kerberos At the time of release, the C++, Java, C#, and Python drivers all provide support for Kerberos authentication to MongoDB. Consider the following tutorials for more information:

- Authenticate to MongoDB with the Java Driver²²
- Authenticate to MongoDB with the C# Driver²³
- Authenticate to MongoDB with the C++ Driver²⁴
- Python Authentication Examples²⁵

Kerberos and the HTTP Console MongoDB does not support kerberizing the [HTTP Console](#)²⁶.

Troubleshooting

Kerberos Configuration Checklist If you're having trouble getting `mongod` to start with Kerberos, there are a number of Kerberos-specific issues that can prevent successful authentication. As you begin troubleshooting your Kerberos deployment, ensure that:

- The `mongod` is from MongoDB Enterprise.
- You are not using the [HTTP Console](#)²⁷. MongoDB Enterprise does not support Kerberos authentication over the HTTP Console interface.
- You have a valid keytab file specified in the environment running the `mongod`. For the `mongod` instance running on the `db0.example.net` host, the service principal should be `mongodb/db0.example.net`.
- DNS allows the `mongod` to resolve the components of the Kerberos infrastructure. You should have both A and PTR records (i.e. forward and reverse DNS) for the system that runs the `mongod` instance.
- The canonical system hostname of the system that runs the `mongod` instance is the resolvable fully qualified domain for this host. Test system hostname resolution with the `hostname -f` command at the system prompt.
- Both the Kerberos KDC and the system running `mongod` instance must be able to resolve each other using DNS²⁸
- The time systems of the systems running the `mongod` instances and the Kerberos infrastructure are synchronized. Time differences greater than 5 minutes will prevent successful authentication.

If you still encounter problems with Kerberos, you can start both `mongod` and `mongo` (or another client) with the environment variable `KRB5_TRACE` set to different files to produce more verbose logging of the Kerberos process to help further troubleshooting, as in the following example:

²²<http://docs.mongodb.org/ecosystem/tutorial/authenticate-with-java-driver/>

²³<http://docs.mongodb.org/ecosystem/tutorial/authenticate-with-csharp-driver/>

²⁴<http://docs.mongodb.org/ecosystem/tutorial/authenticate-with-cpp-driver/>

²⁵<http://api.mongodb.org/python/current/examples/authentication.html>

²⁶<http://docs.mongodb.org/ecosystem/tools/http-interface/#http-console>

²⁷<http://docs.mongodb.org/ecosystem/tools/http-interface/#http-console>

²⁸ By default, Kerberos attempts to resolve hosts using the content of the `/etc/krb5.conf` before using DNS to resolve hosts.

```
env KRB5_KTNAME=/opt/mongodb/mongod.keytab \
KRB5_TRACE=/opt/mongodb/log/mongodb-kerberos.log \
/opt/mongodb/bin/mongod --dbpath /opt/mongodb/data \
--fork --logpath /opt/mongodb/log/mongod.log \
--auth --setParameter authenticationMechanisms=GSSAPI
```

Common Error Messages In some situations, MongoDB will return error messages from the GSSAPI interface if there is a problem with the Kerberos service.

GSSAPI error in client while negotiating security context.

This error occurs on the client and reflects insufficient credentials or a malicious attempt to authenticate.

If you receive this error ensure that you’re using the correct credentials and the correct fully qualified domain name when connecting to the host.

GSSAPI error acquiring credentials.

This error only occurs when attempting to start the mongod or mongos and reflects improper configuration of system hostname or a missing or incorrectly configured keytab file. If you encounter this problem, consider all the items in the [Kerberos Configuration Checklist](#) (page 260), in particular:

- examine the keytab file, with the following command:

```
klist -k <keytab>
```

Replace <keytab> with the path to your keytab file.

- check the configured hostname for your system, with the following command:

```
hostname -f
```

Ensure that this name matches the name in the keytab file, or use the saslHostName to pass MongoDB the correct hostname.

Enable the Traditional MongoDB Authentication Mechanism For testing and development purposes you can enable both the Kerberos (i.e. GSSAPI) authentication mechanism in combination with the traditional MongoDB challenge/response authentication mechanism (i.e. MONGODB-CR), using the following setParameter run-time option:

```
mongod --setParameter authenticationMechanisms=GSSAPI,MONGODB-CR
```

Warning: All keyFile *internal* authentication between members of a *replica set* or *sharded cluster* still uses the MONGODB-CR authentication mechanism, even if MONGODB-CR is not enabled. All client authentication will still use Kerberos.

5.3.3 Create a Vulnerability Report

If you believe you have discovered a vulnerability in MongoDB or have experienced a security incident related to MongoDB, please report the issue so it can be avoided in the future.

To report an issue, we strongly suggest filing a ticket in our ““Security”” project in JIRA <<https://jira.mongodb.org/browse/SECURITY/>>_. MongoDB, Inc responds to vulnerability notifications within 48 hours.

Create the Report in JIRA

Submit a ticket in the [Security²⁹](#) project at: <https://jira.mongodb.org/browse/SECURITY>. The ticket number will become the reference identification for the issue for its lifetime. You can use this identifier for tracking purposes.

Information to Provide

All vulnerability reports should contain as much information as possible so MongoDB's developers can move quickly to resolve the issue. In particular, please include the following:

- The name of the product.
- *Common Vulnerability* information, if applicable, including:
- CVSS (Common Vulnerability Scoring System) Score.
- CVE (Common Vulnerability and Exposures) Identifier.
- Contact information, including an email address and/or phone number, if applicable.

Send the Report via Email

While JIRA is the preferred reporting method, you may also report vulnerabilities via email to [security@mongodb.com³⁰](mailto:security@mongodb.com).

You may encrypt email using MongoDB's public key at <http://docs.mongodb.org/10gen-gpg-key.asc>.

MongoDB, Inc. responds to vulnerability reports sent via email with a response email that contains a reference number for a JIRA ticket posted to the [SECURITY³¹](#) project.

Evaluation of a Vulnerability Report

MongoDB, Inc. validates all submitted vulnerabilities and uses Jira to track all communications regarding a vulnerability, including requests for clarification or additional information. If needed, MongoDB representatives set up a conference call to exchange information regarding the vulnerability.

Disclosure

MongoDB, Inc. requests that you do *not* publicly disclose any information regarding the vulnerability or exploit the issue until it has had the opportunity to analyze the vulnerability, to respond to the notification, and to notify key users, customers, and partners.

The amount of time required to validate a reported vulnerability depends on the complexity and severity of the issue. MongoDB, Inc. takes all required vulnerabilities very seriously and will always ensure that there is a clear and open channel of communication with the reporter.

After validating an issue, MongoDB, Inc. coordinates public disclosure of the issue with the reporter in a mutually agreed timeframe and format. If required or requested, the reporter of a vulnerability will receive credit in the published security bulletin.

²⁹<https://jira.mongodb.org/browse/SECURITY>

³⁰security@mongodb.com

³¹<https://jira.mongodb.org/browse/SECURITY>

5.4 Security Reference

5.4.1 Security Methods in the mongo Shell

Name	Description
<code>db.addUser()</code>	Adds a user to a database, and allows administrators to configure the user's privileges.
<code>db.auth()</code>	Authenticates a user to a database.
<code>db.changeUserPassword()</code>	Changes an existing user's password.

5.4.2 Security Reference Documentation

[User Privilege Roles in MongoDB \(page 263\)](#) Reference on user privilege roles and corresponding access.

[system.users Privilege Documents \(page 268\)](#) Reference on documents used to store user credentials and privilege roles.

[Default MongoDB Port \(page 270\)](#) List of default ports used by MongoDB.

User Privilege Roles in MongoDB

New in version 2.4: In version 2.4, MongoDB adds support for the following user roles:

Roles

Changed in version 2.4.

Roles in MongoDB provide users with a set of specific privileges, on specific logical databases. Users may have multiple roles and may have different roles on different logical database. Roles only grant privileges and never limit access: if a user has `read` (page 263) and `readWriteAnyDatabase` (page 267) permissions on the `records` database, that user will be able to write data to the `records` database.

Note: By default, MongoDB 2.4 is backwards-compatible with the MongoDB 2.2 access control roles. You can explicitly disable this backwards-compatibility by setting the `supportCompatibilityForPrivilegeDocuments` option to 0 during startup, as in the following command-line invocation of MongoDB:

```
mongod --setParameter supportCompatibilityForPrivilegeDocuments=0
```

In general, you should set this option if your deployment does not need to support legacy user documents. Typically legacy user documents are only useful during the upgrade process and while you migrate applications to the updated privilege document form.

See [privilege documents](#) (page 268) and [Delegated Credentials for MongoDB Authentication](#) (page 270) for more information about permissions and authentication in MongoDB.

Database User Roles

`read`

Provides users with the ability to read data from any collection within a specific logical database. This includes `find()` and the following *database commands*:

- `aggregate`

- checkShardingIndex
- cloneCollectionAsCapped
- collStats
- count
- dataSize
- dbHash
- dbStats
- distinct
- filemd5
- geoNear
- geoSearch
- geoWalk
- group
- mapReduce (inline output only.)
- text (beta feature.)

readWrite

Provides users with the ability to read from or write to any collection within a specific logical database. Users with [readWrite](#) (page 264) have access to all of the operations available to [read](#) (page 263) users, as well as the following basic write operations: `insert()`, `remove()`, and `update()`.

Additionally, users with the [readWrite](#) (page 264) have access to the following *database commands*:

- cloneCollection (as the target database.)
- convertToCapped
- create (and to create collections implicitly.)
- drop ()
- dropIndexes
- emptycapped
- ensureIndex ()
- findAndModify
- mapReduce (output to a collection.)
- renameCollection (within the same database.)

Database Administration Roles**dbAdmin**

Provides the ability to perform the following set of administrative operations within the scope of this logical database.

- clean
- collMod
- collStats
- compact

- convertToCapped
- create
- db.createCollection()
- dbStats
- drop()
- dropIndexes
- ensureIndex()
- indexStats
- profile
- reIndex
- renameCollection (within a single database.)
- validate

Furthermore, only `dbAdmin` (page 264) has the ability to read the `system.profile` (page 223) collection.

userAdmin

Allows users to read and write data to the `system.users` (page 268) collection of any database. Users with this role will be able to modify permissions for existing users and create new users. `userAdmin` (page 265) does not restrict the permissions that a user can grant, and a `userAdmin` (page 265) user can grant privileges to themselves or other users in excess of the `userAdmin` (page 265) users' current privileges.

Important: `userAdmin` (page 265) is effectively the *superuser* role for a specific database. Users with `userAdmin` (page 265) can grant themselves all privileges. However, `userAdmin` (page 265) does not explicitly authorize a user for any privileges beyond user administration.

Note: The `userAdmin` (page 265) role is a database-specific privilege, and *only* grants a user the ability to administer users on a single database. However, for the `admin` database, `userAdmin` (page 265) allows a user the ability to gain `userAdminAnyDatabase` (page 267). Thus, for the `admin` database **only**, these roles are effectively the same.

Administrative Roles

clusterAdmin

`clusterAdmin` (page 265) grants access to several administration operations that affect or present information about the whole system, rather than just a single database. These privileges include but are not limited to *replica set* and *sharded cluster* administrative functions.

`clusterAdmin` (page 265) is only applicable on the `admin` database, and does not confer any access to the `local` or `config` databases.

Specifically, users with the `clusterAdmin` (page 265) role have access to the following operations:

- addShard
- closeAllDatabases
- connPoolStats
- connPoolSync
- _cpuProfilerStart

- `_cpuProfilerStop`
- `cursorInfo`
- `diagLogging`
- `dropDatabase`
- `enableSharding`
- `flushRouterConfig`
- `fsync`
- `db.fsyncUnlock()`
- `getCmdLineOpts`
- `getLog`
- `getParameter`
- `getShardMap`
- `getShardVersion`
- `hostInfo`
- `db.currentOp()`
- `db.killOp()`
- `listDatabases`
- `listShards`
- `logRotate`
- `moveChunk`
- `movePrimary`
- `netstat`
- `removeShard`
- `repairDatabase`
- `replSetFreeze`
- `replSetGetStatus`
- `replSetInitiate`
- `replSetMaintenance`
- `replSetReconfig`
- `replSetStepDown`
- `replSetSyncFrom`
- `resync`
- `serverStatus`
- `setParameter`
- `setShardVersion`
- `shardCollection`

- shardingState
- shutdown
- splitChunk
- splitVector
- split
- top
- touch
- unsetSharding

For some cluster administration operations, MongoDB requires read and write access to the `local` or `config` databases. You must specify this access separately from [clusterAdmin](#) (page 265). See the [Combined Access](#) (page 267) section for more information.

Any Database Roles

Note: You must specify the following “any” database roles on the `admin` databases. These roles apply to all databases in a `mongod` instance and are roughly equivalent to their single-database equivalents.

If you add any of these roles to a [user privilege document](#) (page 268) outside of the `admin` database, the privilege will have no effect. However, only the specification of the roles must occur in the `admin` database, with [delegated authentication credentials](#) (page 270), users can gain these privileges by authenticating to another database.

`readAnyDatabase`

[readAnyDatabase](#) (page 267) provides users with the same read-only permissions as [read](#) (page 263), except it applies to *all* logical databases in the MongoDB environment.

`readWriteAnyDatabase`

[readWriteAnyDatabase](#) (page 267) provides users with the same read and write permissions as [readWrite](#) (page 264), except it applies to *all* logical databases in the MongoDB environment.

`userAdminAnyDatabase`

[userAdminAnyDatabase](#) (page 267) provides users with the same access to user administration operations as [userAdmin](#) (page 265), except it applies to *all* logical databases in the MongoDB environment.

Important: Because users with [userAdminAnyDatabase](#) (page 267) and [userAdmin](#) (page 265) have the ability to create and modify permissions in addition to their own level of access, this role is *effectively* the MongoDB system superuser. However, [userAdminAnyDatabase](#) (page 267) and [userAdmin](#) (page 265) do not explicitly authorize a user for any privileges beyond user administration.

`dbAdminAnyDatabase`

[dbAdminAnyDatabase](#) (page 267) provides users with the same access to database administration operations as [dbAdmin](#) (page 264), except it applies to *all* logical databases in the MongoDB environment.

Combined Access

Some operations are only available to users that have multiple roles. Consider the following:

`sh.status()` Requires [clusterAdmin](#) (page 265) and [read](#) (page 263) access to the `config` (page 556) database.

applyOps, **eval**³² Requires [readWriteAnyDatabase](#) (page 267), [userAdminAnyDatabase](#) (page 267), [dbAdminAnyDatabase](#) (page 267) and [clusterAdmin](#) (page 265) (on the admin database.)

Some operations related to cluster administration are not available to users who *only* have the [clusterAdmin](#) (page 265) role:

rs.conf() Requires [read](#) (page 263) on the local database.

sh.addShard() Requires [readWrite](#) (page 264) on the config database.

system.users Privilege Documents

Changed in version 2.4.

Overview

The documents in the `<database>.system.users` (page 268) collection store credentials and user privilege information used by the authentication system to provision access to users in the MongoDB system. See [User Privilege Roles in MongoDB](#) (page 263) for more information about access roles, and [Security](#) (page 233) for an overview of security in MongoDB.

Data Model

`<database>.system.users`

Changed in version 2.4.

Documents in the `<database>.system.users` (page 268) collection stores credentials and [user roles](#) (page 263) for users who have access to the database. Consider the following prototypes of user privilege documents:

```
{  
    user: "<username>",  
    pwd: "<hash>",  
    roles: []  
}  
  
{  
    user: "<username>",  
    userSource: "<database>",  
    roles: []  
}
```

Note: The [pwd](#) (page 269) and [userSource](#) (page 269) fields are mutually exclusive. A single document cannot contain both.

The following privilege document with the [otherDBRoles](#) (page 269) field is only supported on the admin database:

```
{  
    user: "<username>",  
    userSource: "<database>",  
    otherDBRoles: {  
        <database0> : [],  
        <database1> : []  
    },
```

```

    roles: []
}
```

Consider the content of the following `fields` in the `system.users` (page 268) documents:

<database>.system.users.user

`user` (page 269) is a string that identifies each user. Users exist in the context of a single logical database; however, users from one database may obtain access in another database by way of the `otherDBRoles` (page 269) field on the `admin` database, the `userSource` (page 269) field, or the `Any Database Roles` (page 267).

<database>.system.users.pwd

`pwd` (page 269) holds a *hashed* shared secret used to authenticate the `user` (page 269). `pwd` (page 269) field is mutually exclusive with the `userSource` (page 269) field.

<database>.system.users.roles

`roles` (page 269) holds an array of user roles. The available roles are:

- `read` (page 263)
- `readWrite` (page 264)
- `dbAdmin` (page 264)
- `userAdmin` (page 265)
- `clusterAdmin` (page 265)
- `readAnyDatabase` (page 267)
- `readWriteAnyDatabase` (page 267)
- `userAdminAnyDatabase` (page 267)
- `dbAdminAnyDatabase` (page 267)

See [Roles](#) (page 263) for full documentation of all available user roles.

<database>.system.users.userSource

A string that holds the name of the database that contains the credentials for the user. If `userSource` (page 269) is `$external`, then MongoDB will use an external resource, such as Kerberos, for authentication credentials.

Note: In the current release, the only external authentication source is Kerberos, which is only available in MongoDB Enterprise.

Use `userSource` (page 269) to ensure that a single user's authentication credentials are only stored in a single location in a `mongod` instance's data.

A `userSource` (page 269) and `user` (page 269) pair identifies a unique user in a MongoDB system.

admin.system.users.otherDBRoles

A document that holds one or more fields with a name that is the name of a database in the MongoDB instance with a value that holds a list of roles this user has on other databases. Consider the following example:

```
{
  user: "admin",
  userSource: "$external",
  roles: [ "clusterAdmin" ],
  otherDBRoles:
  {
    ...
  }
}
```

```
        config: [ "read" ],
        records: [ "dbAdmin" ]
    }
}
```

This user has the following privileges:

- [clusterAdmin](#) (page 265) on the `admin` database,
- [read](#) (page 263) on the `config` (page 556) database, and
- [dbAdmin](#) (page 264) on the `records` database.

Delegated Credentials for MongoDB Authentication

New in version 2.4.

With a new document format in the `system.users` (page 268) collection, MongoDB now supports the ability to delegate authentication credentials to other sources and databases. The `userSource` (page 269) field in these documents forces MongoDB to use another source for credentials.

Consider the following document in a `system.users` (page 268) collection in a database named `accounts`:

```
{
  user: "application0",
  pwd: "YvuolxMtaycghk2GMrzmImkG4073jzAw2AliMRul",
  roles: []
}
```

Then for *every* database that the `application0` user requires access, add documents to the `system.users` (page 268) collection that resemble the following:

```
{
  user: "application0",
  roles: ['readWrite'],
  userSource: "accounts"
}
```

To gain privileges to databases where the `application0` has access, you must first authenticate to the `accounts` database.

Disable Legacy Privilege Documents

By default MongoDB 2.4 includes support for both new, role-based privilege documents style as well 2.2 and earlier privilege documents. MongoDB assumes any privilege document without a `roles` (page 269) field is a 2.2 or earlier document.

To ensure that `mongod` instances will only provide access to users defined with the new role-based privilege documents, use the following `setParameter` run-time option:

```
mongod --setParameter supportCompatibilityForPrivilegeDocuments=0
```

Default MongoDB Port

The following table lists the default ports used by MongoDB:

Default Port	Description
27017	The default port for mongod and mongos instances. You can change this port with <code>port</code> or <code>--port</code> .
27018	The default port when running with <code>--shardsvr</code> runtime operation or <code>shardsvr</code> setting.
27019	The default port when running with <code>--configsvr</code> runtime operation or <code>configsvr</code> setting.
28017	The default port for the web status page. The web status page is always accessible at a port number that is 1000 greater than the port determined by <code>port</code> .

5.4.3 Security Release Notes Alerts

Security Release Notes (page 271) Security vulnerability for password.

Security Release Notes

Access to `system.users` Collection

Changed in version 2.4.

In 2.4, only users with the `userAdmin` role have access to the `system.users` collection.

In version 2.2 and earlier, the read-write users of a database all have access to the `system.users` collection, which contains the user names and user password hashes.³³

Password Hashing Insecurity

If a user has the same password for multiple databases, the hash will be the same. A malicious user could exploit this to gain access on a second database using a different user's credentials.

As a result, always use unique username and password combinations for each database.

Thanks to Will Urbanski, from Dell SecureWorks, for identifying this issue.

³³ Read-only users do not have access to the `system.users` collection.

Aggregation

Aggregations operations process data records and return computed results. Aggregation operations group values from multiple documents together, and can perform a variety of operations on the grouped data to return a single result. MongoDB provides three ways to perform aggregation: the [aggregation pipeline](#) (page 277), the [map-reduce function](#) (page 280), and [single purpose aggregation methods and commands](#) (page 282).

Aggregation Introduction (page 273) A high-level introduction to aggregation.

Aggregation Concepts (page 277) Introduces the use and operation of the data aggregation modalities available in MongoDB.

Aggregation Pipeline (page 277) The aggregation pipeline is a framework for performing aggregation tasks, modeled on the concept of data processing pipelines. Using this framework, MongoDB passes the documents of a single collection through a pipeline. The pipeline transforms the documents into aggregated results, and is accessed through the `aggregate` database command.

Map-Reduce (page 280) Map-reduce is a generic multi-phase data aggregation modality for processing quantities of data. MongoDB provides map-reduce with the `mapReduce` database command.

Single Purpose Aggregation Operations (page 282) MongoDB provides a collection of specific data aggregation operations to support a number of common data aggregation functions. These operations include returning counts of documents, distinct values of a field, and simple grouping operations.

Aggregation Mechanics (page 285) Details internal optimization operations, limits, support for sharded collections, and concurrency concerns.

Aggregation Examples (page 288) Examples and tutorials for data aggregation operations in MongoDB.

Aggregation Reference (page 304) References for all aggregation operations material for all data aggregation methods in MongoDB.

6.1 Aggregation Introduction

Aggregations are operations that process data records and return computed results. MongoDB provides a rich set of aggregation operations that examine and perform calculations on the data sets. Running data aggregation on the `mongod` instance simplifies application code and limits resource requirements.

Like queries, aggregation operations in MongoDB use *collections* of documents as an input and return results in the form of one or more documents.

6.1.1 Aggregation Modalities

Aggregation Pipelines

MongoDB 2.2 introduced a new *aggregation framework* (page 277), modeled on the concept of data processing pipelines. Documents enter a multi-stage pipeline that transforms the documents into an aggregated result.

The most basic pipeline stages provide *filters* that operate like queries and *document transformations* that modify the form of the output document.

Other pipeline operations provide tools for grouping and sorting documents by specific field or fields as well as tools for aggregating the contents of arrays, including arrays of documents. In addition, pipeline stages can use *operators* for tasks such as calculating the average or concatenating a string.

The pipeline provides efficient data aggregation using native operations within MongoDB, and is the preferred method for data aggregation in MongoDB.

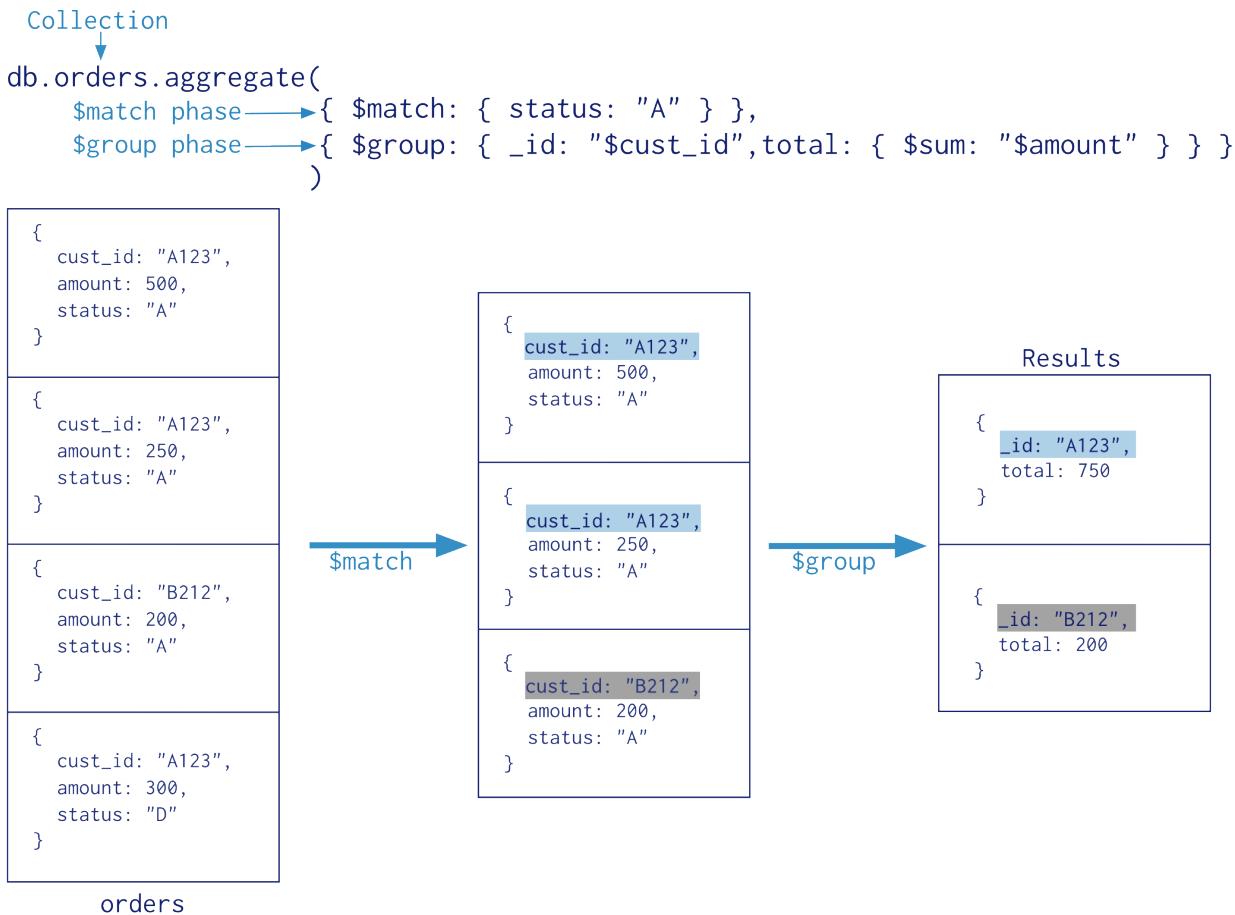


Figure 6.1: Diagram of the annotated aggregation pipeline operation. The aggregation pipeline has two phases: `$match` and `$group`.

Map-Reduce

MongoDB also provides *map-reduce* (page 280) operations to perform aggregation. In general, map-reduce operations have two phases: a *map* stage that processes each document and *emits* one or more objects for each input document,

and *reduce* phase that combines the output of the map operation. Optionally, map-reduce can have a *finalize* stage to make final modifications to the result. Like other aggregation operations, map-reduce can specify a query condition to select the input documents as well as sort and limit the results.

Map-reduce uses custom JavaScript functions to perform the map and reduce operations, as well as the optional *finalize* operation. While the custom JavaScript provide great flexibility compared to the aggregation pipeline, in general, map-reduce is less efficient and more complex than the aggregation pipeline.

Additionally, map-reduce operations can have output sets that exceed the 16 megabyte output limitation of the aggregation pipeline.

Note: Starting in MongoDB 2.4, certain mongo shell functions and properties are inaccessible in map-reduce operations. MongoDB 2.4 also provides support for multiple JavaScript operations to run at the same time. Before MongoDB 2.4, JavaScript code executed in a single thread, raising concurrency issues for map-reduce.

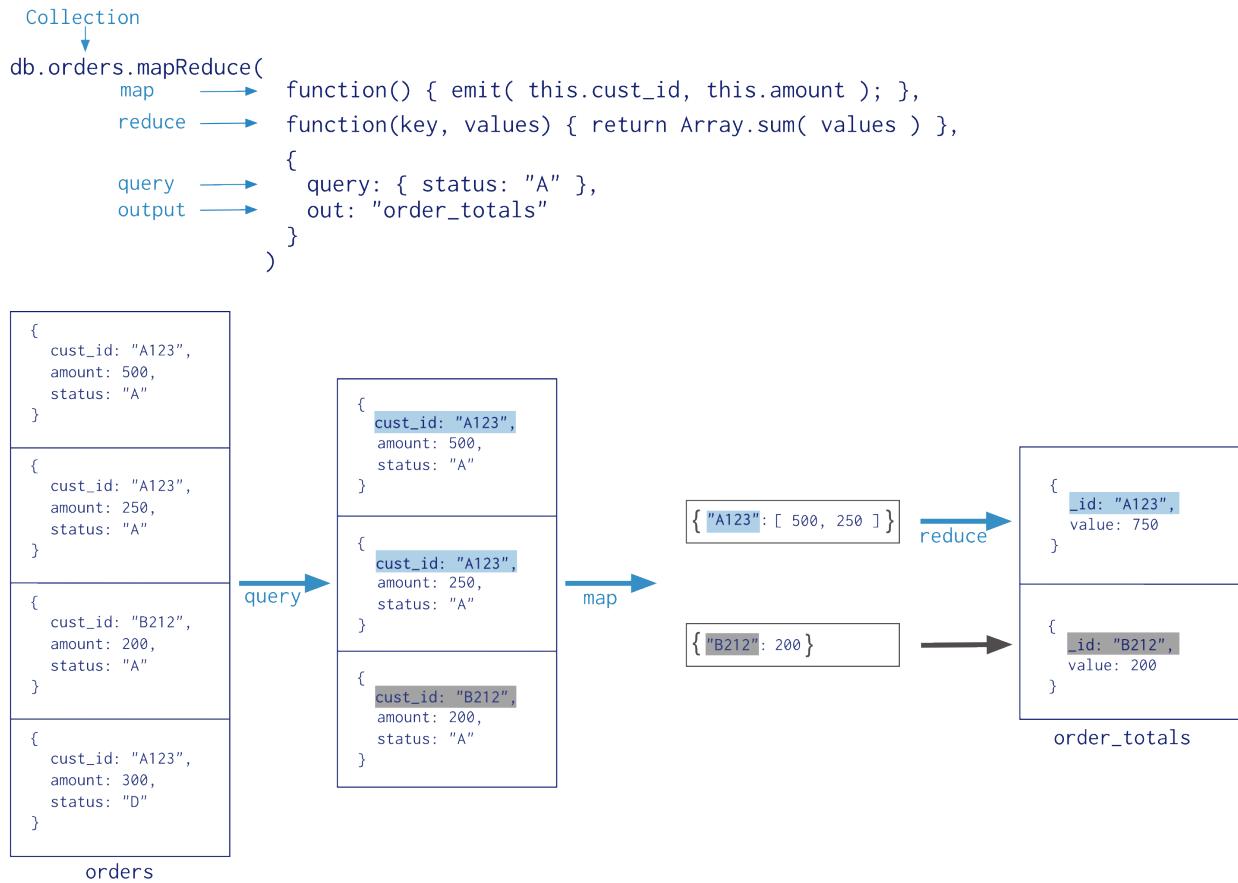


Figure 6.2: Diagram of the annotated map-reduce operation.

Single Purpose Aggregation Operations

For a number of common *single purpose aggregation operations* (page 282), MongoDB provides special purpose database commands. These common aggregation operations are: returning a count of matching documents, returning the distinct values for a field, and grouping data based on the values of a field. All of these operations aggregate documents from a single collection. While these operations provide simple access to common aggregation processes, they lack the flexibility and capabilities of the aggregation pipeline and map-reduce.

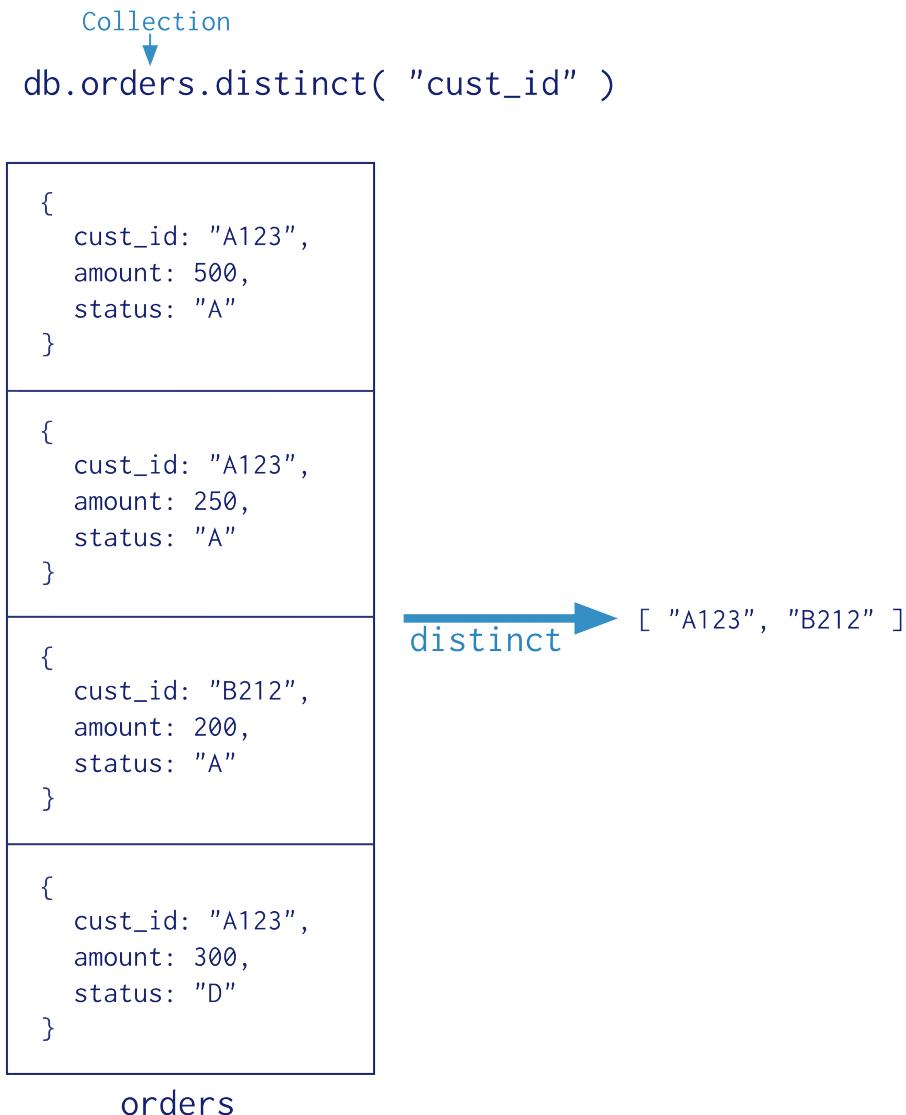


Figure 6.3: Diagram of the annotated distinct operation.

6.1.2 Additional Features and Behaviors

Both the aggregation pipeline and map-reduce can operate on a [sharded collection](#) (page 487). Map-reduce operations can also output to a sharded collection. See [Aggregation Pipeline and Sharded Collections](#) (page 287) and [Map-Reduce and Sharded Collections](#) (page 287) for details.

The aggregation pipeline can use indexes to improve its performance during some of its stages. In addition, the aggregation pipeline has an internal optimization phase. See [Pipeline Operators and Indexes](#) (page 279) and [Aggregation Pipeline Optimization](#) (page 285) for details.

For a feature comparison of the aggregation pipeline, map-reduce, and the special group functionality, see [Aggregation Commands Comparison](#) (page 304).

6.2 Aggregation Concepts

MongoDB provides the three approaches to aggregation, each with its own strengths and purposes for a given situation. This section describes these approaches and also describes behaviors and limitations specific to each approach. See also the [chart](#) (page 304) that compares the approaches.

Aggregation Pipeline (page 277) The aggregation pipeline is a framework for performing aggregation tasks, modeled on the concept of data processing pipelines. Using this framework, MongoDB passes the documents of a single collection through a pipeline. The pipeline transforms the documents into aggregated results, and is accessed through the `aggregate` database command.

Map-Reduce (page 280) Map-reduce is a generic multi-phase data aggregation modality for processing quantities of data. MongoDB provides map-reduce with the `mapReduce` database command.

Single Purpose Aggregation Operations (page 282) MongoDB provides a collection of specific data aggregation operations to support a number of common data aggregation functions. These operations include returning counts of documents, distinct values of a field, and simple grouping operations.

Aggregation Mechanics (page 285) Details internal optimization operations, limits, support for sharded collections, and concurrency concerns.

6.2.1 Aggregation Pipeline

New in version 2.2.

The aggregation pipeline is a framework for data aggregation modeled on the concept of data processing pipelines. Documents enter a multi-stage pipeline that transforms the documents into an aggregated results.

The aggregation pipeline provides an alternative to `map-reduce` and may be the preferred solution for many aggregation tasks where the complexity of map-reduce may be unwarranted.

Aggregation pipeline have some limitations on value types and result size. See [Aggregation Pipeline Limits](#) (page 286) for details on limits and restrictions on the aggregation pipeline.

Pipeline

Conceptually, documents from a collection travel through an aggregation pipeline, which transforms these objects as they pass through. For those familiar with UNIX-like shells (e.g. bash,) the concept is analogous to the pipe (i.e. `|`).

The MongoDB aggregation pipeline starts with the documents of a collection and streams the documents from one *pipeline operator* to the next to process the documents. Each operator in the pipeline transforms the documents as they pass through the pipeline. Pipeline operators do not need to produce one output document for every input document. Operators may generate new documents or filter out documents. Pipeline operators can be repeated in the pipeline.

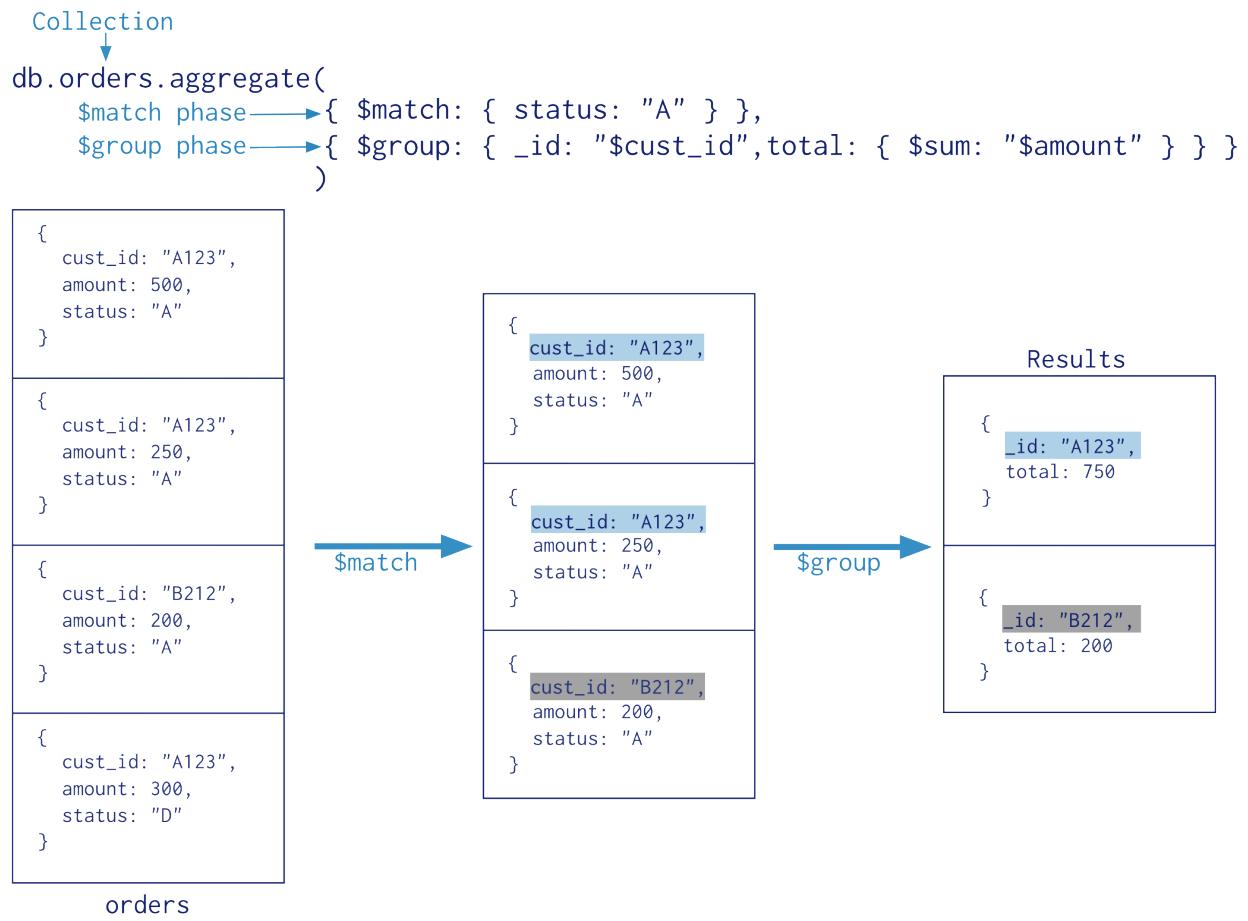


Figure 6.4: Diagram of the annotated aggregation pipeline operation. The aggregation pipeline has two phases: `$match` and `$group`.

Important: The result of aggregation pipeline is a *document* and is subject to the *BSON Document size* limit, which is currently 16 megabytes.

See [aggregation-pipeline-operator-reference](#) for the list of pipeline operators that define the stages.

For example usage of the aggregation pipeline, consider [Aggregation with User Preference Data](#) (page 292) and [Aggregation with the Zip Code Data Set](#) (page 289), as well as the `aggregate` command and the `db.collection.aggregate()` method reference pages.

Pipeline Expressions

Each pipeline operator takes a pipeline expression as its operand. Pipeline expressions specify the transformation to apply to the input documents. Expressions have a *document* structure and can contain fields, values, and *operators*.

Pipeline expressions can only operate on the current document in the pipeline and cannot refer to data from other documents: expression operations provide in-memory transformation of documents.

Generally, expressions are stateless and are only evaluated when seen by the aggregation process with one exception: *accumulator* expressions. The accumulator expressions, used with the `$group` pipeline operator, maintain their state (e.g. totals, maximums, minimums, and related data) as documents progress through the pipeline.

For the expression operators, see [aggregation-expression-operators](#).

Aggregation Pipeline Behavior

In MongoDB, the `aggregate` command operates on a single collection, logically passing the *entire* collection into the aggregation pipeline. To optimize the operation, wherever possible, use the following strategies to avoid scanning the entire collection.

Pipeline Operators and Indexes

The `$match`, `$sort`, `$limit`, and `$skip` pipeline operators can take advantage of an index when they occur at the **beginning** of the pipeline **before** any of the following aggregation operators: `$project`, `$unwind`, and `$group`.

New in version 2.4: The `$geoNear` pipeline operator takes advantage of a geospatial index. When using `$geoNear`, the `$geoNear` pipeline operation must appear as the first stage in an aggregation pipeline.

For unsharded collections, when the aggregation pipeline only needs to access the indexed fields to fulfill its operations, an index can [cover](#) (page 35) the pipeline.

Example

Consider the following index on the `orders` collection:

```
{ status: 1, amount: 1, cust_id: 1 }
```

This index can cover the following aggregation pipeline operation because MongoDB does not need to inspect the data outside of the index to fulfill the operation:

```
db.orders.aggregate([
    { $match: { status: "A" } },
    { $group: { _id: "$cust_id", total: { $sum: "$amount" } } },
    { $sort: { total: -1 } }
])
```

Early Filtering

If your aggregation operation requires only a subset of the data in a collection, use the `$match`, `$limit`, and `$skip` stages to restrict the documents that enter at the beginning of the pipeline. When placed at the beginning of a pipeline, `$match` operations use suitable indexes to scan only the matching documents in a collection.

Placing a `$match` pipeline stage followed by a `$sort` stage at the start of the pipeline is logically equivalent to a single query with a sort and can use an index. When possible, place `$match` operators at the beginning of the pipeline.

Additional Features

The aggregation pipeline has an internal optimization phase that provides improved performance for certain sequences of operators. For details, see [Pipeline Sequence Optimization](#) (page 285).

The aggregation pipeline supports operations on sharded collections. See [Aggregation Pipeline and Sharded Collections](#) (page 287).

6.2.2 Map-Reduce

Map-reduce is a data processing paradigm for condensing large volumes of data into useful *aggregated* results. For map-reduce operations, MongoDB provides the `mapReduce` database command.

Consider the following map-reduce operation:

In this map-reduce operation, MongoDB applies the *map* phase to each input document (i.e. the documents in the collection that match the query condition). The map function emits key-value pairs. For those keys that have multiple values, MongoDB applies the *reduce* phase, which collects and condenses the aggregated data. MongoDB then stores the results in a collection. Optionally, the output of the reduce function may pass through a *finalize* function to further condense or process the results of the aggregation.

All map-reduce functions in MongoDB are JavaScript and run within the `mongod` process. Map-reduce operations take the documents of a single *collection* as the *input* and can perform any arbitrary sorting and limiting before beginning the map stage. `mapReduce` can return the results of a map-reduce operation as a document, or may write the results to collections. The input and the output collections may be sharded.

Note: For most aggregation operations, the [Aggregation Pipeline](#) (page 277) provides better performance and more coherent interface. However, map-reduce operations provide some flexibility that is not presently available in the aggregation pipeline.

Map-Reduce JavaScript Functions

In MongoDB, map-reduce operations use custom JavaScript functions to *map*, or associate, values to a key. If a key has multiple values mapped to it, the operation *reduces* the values for the key to a single object.

The use of custom JavaScript functions provide flexibility to map-reduce operations. For instance, when processing a document, the map function can create more than one key and value mapping or no mapping. Map-reduce operations can also use a custom JavaScript function to make final modifications to the results at the end of the map and reduce operation, such as perform additional calculations.

Map-Reduce Behavior

In MongoDB, the map-reduce operation can write results to a collection or return the results inline. If you write map-reduce output to a collection, you can perform subsequent map-reduce operations on the same input collection

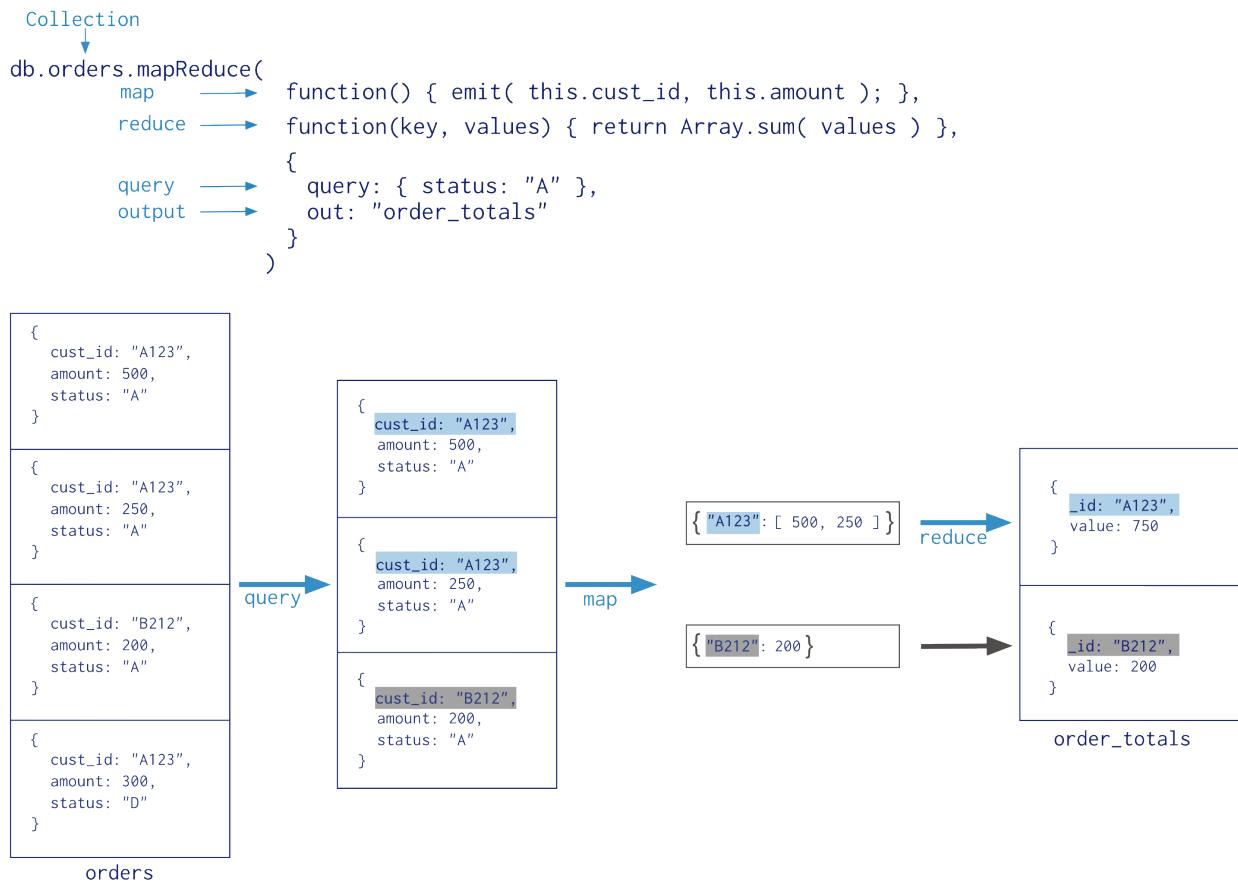


Figure 6.5: Diagram of the annotated map-reduce operation.

that merge replace, merge, or reduce new results with previous results. See `mapReduce` and [Perform Incremental Map-Reduce](#) (page 298) for details and examples.

When returning the results of a map reduce operation *inline*, the result documents must be within the BSON Document Size limit, which is currently 16 megabytes. For additional information on limits and restrictions on map-reduce operations, see the <http://docs.mongodb.org/manual/reference/command/mapReduce> reference page.

MongoDB supports map-reduce operations on [sharded collections](#) (page 487). Map-reduce operations can also output the results to a sharded collection. See [Map-Reduce and Sharded Collections](#) (page 287).

6.2.3 Single Purpose Aggregation Operations

Aggregation refers to a broad class of data manipulation operations that compute a result based on an input *and* a specific procedure. MongoDB provides a number of aggregation operations that perform specific aggregation operations on a set of data.

Although limited in scope, particularly compared to the [aggregation pipeline](#) (page 277) and [map-reduce](#) (page 280), these operations provide straightforward semantics for common data processing options.

Count

MongoDB can return a count of the number of documents that match a query. The `count` command as well as the `count()` and `cursor.count()` methods provide access to counts in the mongo shell.

Example

Given a collection named `records` with *only* the following documents:

```
{ a: 1, b: 0 }
{ a: 1, b: 1 }
{ a: 1, b: 4 }
{ a: 2, b: 2 }
```

The following operation would count all documents in the collection and return the number 4:

```
db.records.count()
```

The following operation will count only the documents where the value of the field `a` is 1 and return 3:

```
db.records.count( { a: 1 } )
```

Distinct

The *distinct* operation takes a number of documents that match a query and returns all of the unique values for a field in the matching documents. The `distinct` command and `db.collection.distinct()` method provide this operation in the mongo shell. Consider the following examples of a distinct operation:

Example

Given a collection named `records` with *only* the following documents:

```
{ a: 1, b: 0 }
{ a: 1, b: 1 }
{ a: 1, b: 1 }
{ a: 1, b: 4 }
```

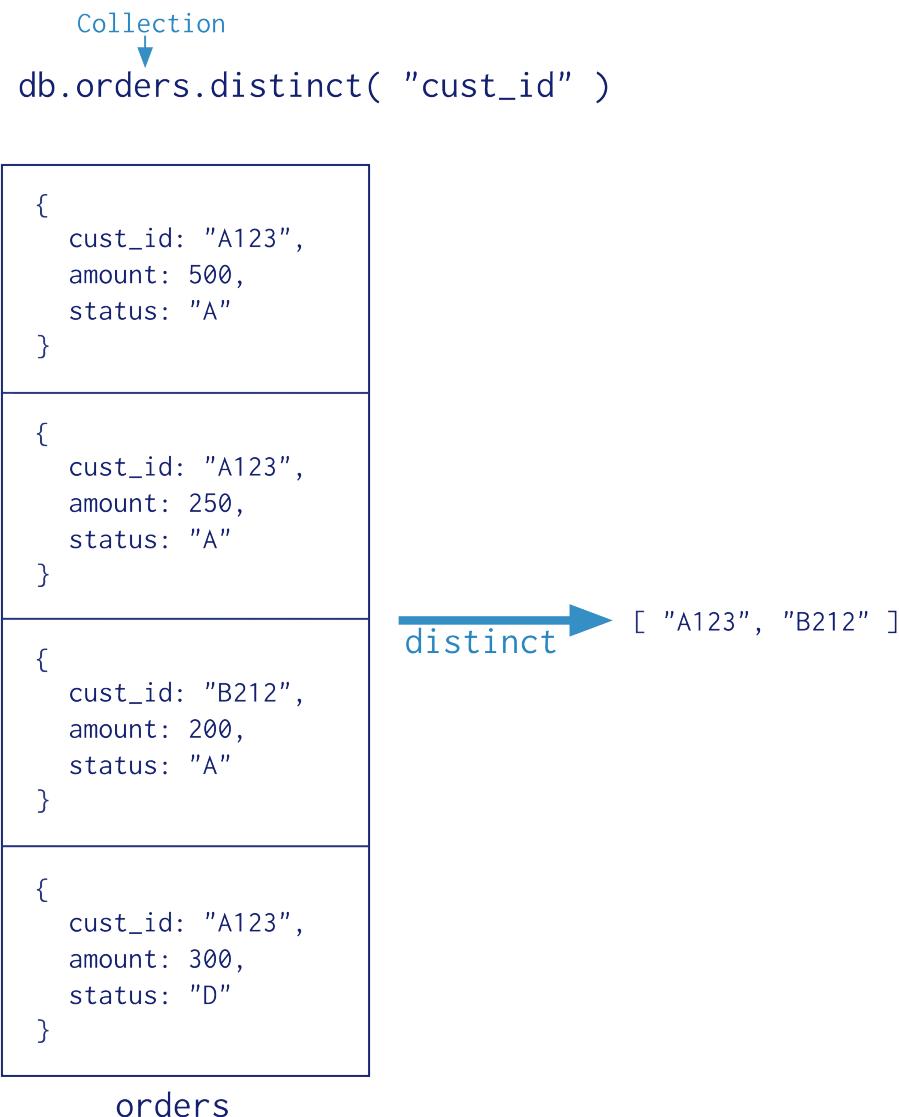


Figure 6.6: Diagram of the annotated distinct operation.

```
{ a: 2, b: 2 }
{ a: 2, b: 2 }
```

Consider the following `db.collection.distinct()` operation which returns the distinct values of the field `b`:

```
db.records.distinct( "b" )
```

The results of this operation would resemble:

```
[ 0, 1, 4, 2 ]
```

Group

The `group` operation takes a number of documents that match a query, and then collects groups of documents based on the value of a field or fields. It returns an array of documents with computed results for each group of documents.

Access the grouping functionality via the `group` command or the `db.collection.group()` method in the mongo shell.

Warning: `group` does not support data in sharded collections. In addition, the results of the `group` operation must be no larger than 16 megabytes.

Consider the following group operation:

Example

Given a collection named `records` with the following documents:

```
{ a: 1, count: 4 }
{ a: 1, count: 2 }
{ a: 1, count: 4 }
{ a: 2, count: 3 }
{ a: 2, count: 1 }
{ a: 1, count: 5 }
{ a: 4, count: 4 }
```

Consider the following group operation which groups documents by the field `a`, where `a` is less than 3, and sums the field `count` for each group:

```
db.records.group( {
  key: { a: 1 },
  cond: { a: { $lt: 3 } },
  reduce: function(cur, result) { result.count += cur.count },
  initial: { count: 0 }
} )
```

The results of this group operation would resemble the following:

```
[
  { a: 1, count: 15 },
  { a: 2, count: 4 }
]
```

See also:

The `$group` for related functionality in the [aggregation pipeline](#) (page 277).

6.2.4 Aggregation Mechanics

This section describes behaviors and limitations for the various aggregation modalities.

[Aggregation Pipeline Optimization \(page 285\)](#) Details the internal optimization of certain pipeline sequence.

[Aggregation Pipeline Limits \(page 286\)](#) Presents limitations on aggregation pipeline operations.

[Aggregation Pipeline and Sharded Collections \(page 287\)](#) Mechanics of aggregation pipeline operations on sharded collections.

[Map-Reduce and Sharded Collections \(page 287\)](#) Mechanics of map-reduce operation with sharded collections.

[Map Reduce Concurrency \(page 288\)](#) Details the locks taken during map-reduce operations.

Aggregation Pipeline Optimization

Changed in version 2.4.

Aggregation pipeline operations have an optimization phase which attempts to rearrange the pipeline for improved performance.

Pipeline Sequence Optimization

`$sort + $skip + $limit` Sequence Optimization When you have a sequence with `$sort` followed by a `$skip` followed by a `$limit`, an optimization occurs that moves the `$limit` operator before the `$skip` operator. For example, if the pipeline consists of the following stages:

```
{ $sort: { age : -1 } },
{ $skip: 10 },
{ $limit: 5 }
```

During the optimization phase, the optimizer transforms the sequence to the following:

```
{ $sort: { age : -1 } },
{ $limit: 15 }
{ $skip: 10 }
```

Note: The `$limit` value has increased to the sum of the initial value and the `$skip` value.

The optimized sequence now has `$sort` immediately preceding the `$limit`. See `$sort` for information on the behavior of the `$sort` operation when it immediately precedes `$limit`.

`$limit + $skip + $limit + $skip` Sequence Optimization When you have a continuous sequence of a `$limit` pipeline stage followed by a `$skip` pipeline stage, the optimization phase attempts to arrange the pipeline stages to combine the limits and skips. For example, if the pipeline consists of the following stages:

```
{ $limit: 100 },
{ $skip: 5 },
{ $limit: 10 },
{ $skip: 2 }
```

During the intermediate step, the optimizer reverses the position of the `$skip` followed by a `$limit` to `$limit` followed by the `$skip`.

```
{ $limit: 100 },
{ $limit: 15},
{ $skip: 5 },
{ $skip: 2 }
```

The `$limit` value has increased to the sum of the initial value and the `$skip` value. Then, for the final `$limit` value, the optimizer selects the minimum between the adjacent `$limit` values. For the final `$skip` value, the optimizer adds the adjacent `$skip` values, to transform the sequence to the following:

```
{ $limit: 15 },
{ $skip: 7 }
```

Projection Optimization

If the aggregation pipeline contains a `$project` stage that specifies the fields to **include**, then MongoDB applies the projection to the head of the pipeline. This reduces the amount of data passing through the pipeline from the start. In the following example, the `$project` stage specifies that the results of this stage return only the `_id` and the `amount` fields. The optimization phase applies the projection to the head of the pipeline such that only the `_id` and the `amount` fields return in the resulting documents from the `$match` stage as well.

```
db.orders.aggregate(
  { $match: { status: "A" } },
  { $project: { amount: 1 } }
)
```

Aggregation Pipeline Limits

Aggregation operations with the `aggregate` command have the following limitations.

Type Restrictions

The [aggregation pipeline](#) (page 277) cannot operate on values of the following types: `Symbol`, `MinKey`, `MaxKey`, `DBRef`, `Code`, `CodeWScope`.

Changed in version 2.4: Removed restriction on `Binary` type data. In MongoDB 2.2, the pipeline could not operate on `Binary` type data.

Result Size Restrictions

Output from the pipeline cannot exceed the `BSON Document Size` limit, which is currently 16 megabytes. If the result set exceeds this limit, the `aggregate` command produces an error.

Memory Restrictions

If any single aggregation operation consumes more than 10 percent of system RAM, the operation will produce an error.

Cumulative operators, such as `$sort` and `$group`, require access to the entire input set before they can produce any output. These operators log a *warning* if the cumulative operator consumes 5% or more of the physical memory on the host. Like any aggregation operation, these operators produce an error if they consume 10% or more of the physical memory on the host. See the `$sort` and `$group` reference pages for details on their specific memory requirements and use.

Aggregation Pipeline and Sharded Collections

The aggregation pipeline supports operations on *sharded* collections. This section describes behaviors specific to the [aggregation pipeline](#) (page 277) and sharded collections.

Note: Changed in version 2.2: Some aggregation pipeline operations will cause mongos instances to require more CPU resources than in previous versions. This modified performance profile may dictate alternate architectural decisions if you use the [aggregation pipeline](#) (page 277) extensively in a sharded environment.

When operating on a sharded collection, the aggregation pipeline is split into two parts. First, the aggregation pipeline pushes all of the operators up to the first \$group or \$sort operation to each shard¹. Then, a second pipeline runs on the mongos. This pipeline consists of the first \$group or \$sort and any remaining pipeline operators, and runs on the results received from the shards.

The \$group operator brings in any “sub-totals” from the shards and combines them: in some cases these may be structures. For example, the \$avg expression maintains a total and count for each shard; mongos combines these values and then divides.

Map-Reduce and Sharded Collections

Map-reduce supports operations on sharded collections, both as an input and as an output. This section describes the behaviors of mapReduce specific to sharded collections.

Sharded Collection as Input

When using sharded collection as the input for a map-reduce operation, mongos will automatically dispatch the map-reduce job to each shard in parallel. There is no special option required. mongos will wait for jobs on all shards to finish.

Sharded Collection as Output

Changed in version 2.2.

If the out field for mapReduce has the sharded value, MongoDB shards the output collection using the _id field as the shard key.

To output to a sharded collection:

- If the output collection does not exist, MongoDB creates and shards the collection on the _id field.
- For a new or an empty sharded collection, MongoDB uses the results of the first stage of the map-reduce operation to create the initial *chunks* distributed among the shards.
- mongos dispatches, in parallel, a map-reduce post-processing job to every shard that owns a chunk. During the post-processing, each shard will pull the results for its own chunks from the other shards, run the final reduce/finalize, and write locally to the output collection.

Note:

- During later map-reduce jobs, MongoDB splits chunks as needed.
- Balancing of chunks for the output collection is automatically prevented during post-processing to avoid consistency issues.

¹ If an early \$match can exclude shards through the use of the shard key in the predicate, then these operators are only pushed to the relevant shards.

In MongoDB 2.0:

- mongos retrieves the results from each shard, performs a merge sort to order the results, and proceeds to the reduce/finalize phase as needed. mongos then writes the result to the output collection in sharded mode.
- This model requires only a small amount of memory, even for large data sets.
- Shard chunks are not automatically split during insertion. This requires manual intervention until the chunks are granular and balanced.

Important: For best results, only use the sharded output options for `mapReduce` in version 2.2 or later.

Map Reduce Concurrency

The map-reduce operation is composed of many tasks, including reads from the input collection, executions of the `map` function, executions of the `reduce` function, writes to a temporary collection during processing, and writes to the output collection.

During the operation, map-reduce takes the following locks:

- The read phase takes a read lock. It yields every 100 documents.
- The insert into the temporary collection takes a write lock for a single write.
- If the output collection does not exist, the creation of the output collection takes a write lock.
- If the output collection exists, then the output actions (i.e. `merge`, `replace`, `reduce`) take a write lock.

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, JavaScript code (i.e. `map`, `reduce`, `finalize` functions) executed in a single thread.

Note: The final write lock during post-processing makes the results appear atomically. However, output actions `merge` and `reduce` may take minutes to process. For the `merge` and `reduce`, the `nonAtomic` flag is available. See the `db.collection.mapReduce()` reference for more information.

6.3 Aggregation Examples

This document provides the practical examples that display the capabilities of [aggregation](#) (page 277).

[**Aggregation with the Zip Code Data Set**](#) (page 289) Use the aggregation pipeline to group values and to calculate aggregated sums and averages for a collection of United States zip codes.

[**Aggregation with User Preference Data**](#) (page 292) Use the pipeline to sort, normalize, and sum data on a collection of user data.

[**Map-Reduce Examples**](#) (page 296) Define map-reduce operations that select ranges, group data, and calculate sums and averages.

[**Perform Incremental Map-Reduce**](#) (page 298) Run a map-reduce operations over one collection and output results to another collection.

[**Troubleshoot the Map Function**](#) (page 300) Steps to troubleshoot the `map` function.

[**Troubleshoot the Reduce Function**](#) (page 301) Steps to troubleshoot the `reduce` function.

6.3.1 Aggregation with the Zip Code Data Set

The examples in this document use the `zipcode` collection. This collection is available at: media.mongodb.org/zips.json². Use `mongoimport` to load this data set into your `mongod` instance.

Data Model

Each document in the `zipcode` collection has the following form:

```
{
  "_id": "10280",
  "city": "NEW YORK",
  "state": "NY",
  "pop": 5574,
  "loc": [
    -74.016323,
    40.710537
  ]
}
```

The `_id` field holds the zip code as a string.

The `city` field holds the city.

The `state` field holds the two letter state abbreviation.

The `pop` field holds the population.

The `loc` field holds the location as a latitude longitude pair.

All of the following examples use the `aggregate()` helper in the mongo shell. `aggregate()` provides a wrapper around the `aggregate` database command. See the documentation for your [driver](#) (page 92) for a more idiomatic interface for data aggregation operations.

Return States with Populations above 10 Million

To return all states with a population greater than 10 million, use the following aggregation operation:

```
db.zipcodes.aggregate( { $group :
  { _id : "$state",
    totalPop : { $sum : "$pop" } } },
  { $match : {totalPop : { $gte : 10*1000*1000 } } } )
```

Aggregations operations using the `aggregate()` helper process all documents in the `zipcodes` collection. `aggregate()` connects a number of [pipeline](#) (page 277) operators, which define the aggregation process.

In this example, the pipeline passes all documents in the `zipcodes` collection through the following steps:

- the `$group` operator collects all documents and creates documents for each state.

These new per-state documents have one field in addition the `_id` field: `totalPop` which is a generated field using the `$sum` operation to calculate the total value of all `pop` fields in the source documents.

After the `$group` operation the documents in the pipeline resemble the following:

²<http://media.mongodb.org/zips.json>

```
{  
  "_id" : "AK",  
  "totalPop" : 550043  
}
```

- the `$match` operation filters these documents so that the only documents that remain are those where the value of `totalPop` is greater than or equal to 10 million.

The `$match` operation does not alter the documents, which have the same format as the documents output by `$group`.

The equivalent *SQL* for this operation is:

```
SELECT state, SUM(pop) AS totalPop  
  FROM zipcodes  
 GROUP BY state  
 HAVING totalPop >= (10*1000*1000)
```

Return Average City Population by State

To return the average populations for cities in each state, use the following aggregation operation:

```
db.zipcodes.aggregate( { $group :  
    { _id : { state : "$state", city : "$city" },  
      pop : { $sum : "$pop" } } },  
  { $group :  
    { _id : "$_id.state",  
      avgCityPop : { $avg : "$pop" } } } )
```

Aggregations operations using the `aggregate()` helper process all documents in the `zipcodes` collection. `aggregate()` connects a number of [pipeline](#) (page 277) operators that define the aggregation process.

In this example, the pipeline passes all documents in the `zipcodes` collection through the following steps:

- the `$group` operator collects all documents and creates new documents for every combination of the `city` and `state` fields in the source document.

After this stage in the pipeline, the documents resemble the following:

```
{  
  "_id" : {  
    "state" : "CO",  
    "city" : "EDGEWATER"  
  },  
  "pop" : 13154  
}
```

- the second `$group` operator collects documents by the `state` field and use the `$avg` expression to compute a value for the `avgCityPop` field.

The final output of this aggregation operation is:

```
{  
  "_id" : "MN",  
  "avgCityPop" : 5335  
},
```

Return Largest and Smallest Cities by State

To return the smallest and largest cities by population for each state, use the following aggregation operation:

```
db.zipcodes.aggregate( { $group:
    { _id: { state: "$state", city: "$city" },
      pop: { $sum: "$pop" } },
    { $sort: { pop: 1 } },
    { $group:
        { _id : "$_id.state",
          biggestCity: { $last: "$_id.city" },
          biggestPop: { $last: "$pop" },
          smallestCity: { $first: "$_id.city" },
          smallestPop: { $first: "$pop" } },
        { $project:
            { _id: 0,
              state: "$_id",
              biggestCity: { name: "$biggestCity", pop: "$biggestPop" },
              smallestCity: { name: "$smallestCity", pop: "$smallestPop" } } } } }
```

Aggregation operations using the `aggregate()` helper process all documents in the `zipcodes` collection. `aggregate()` combines a number of [pipeline](#) (page 277) operators that define the aggregation process.

All documents from the `zipcodes` collection pass into the pipeline, which consists of the following steps:

- the `$group` operator collects all documents and creates new documents for every combination of the `city` and `state` fields in the source documents.

By specifying the value of `_id` as a sub-document that contains both fields, the operation preserves the `state` field for use later in the pipeline. The documents produced by this stage of the pipeline have a second field, `pop`, which uses the `$sum` operator to provide the total of the `pop` fields in the source document.

At this stage in the pipeline, the documents resemble the following:

```
{
  "_id" : {
    "state" : "CO",
    "city" : "EDGEWATER"
  },
  "pop" : 13154
}
```

- `$sort` operator orders the documents in the pipeline based on the vale of the `pop` field from largest to smallest. This operation does not alter the documents.
- the second `$group` operator collects the documents in the pipeline by the `state` field, which is a field inside the nested `_id` document.

Within each per-state document this `$group` operator specifies four fields: Using the `$last` expression, the `$group` operator creates the `biggestcity` and `biggestpop` fields that store the city with the largest population and that population. Using the `$first` expression, the `$group` operator creates the `smallestcity` and `smallestpop` fields that store the city with the smallest population and that population.

The documents, at this stage in the pipeline resemble the following:

```
{
  "_id" : "WA",
```

```
        "biggestCity" : "SEATTLE",
        "biggestPop" : 520096,
        "smallestCity" : "BENGE",
        "smallestPop" : 2
    }
```

- The final operation is `$project`, which renames the `_id` field to `state` and moves the `biggestCity`, `biggestPop`, `smallestCity`, and `smallestPop` into `biggestCity` and `smallestCity` sub-documents.

The output of this aggregation operation is:

```
{
  "state" : "RI",
  "biggestCity" : {
    "name" : "CRANSTON",
    "pop" : 176404
  },
  "smallestCity" : {
    "name" : "CLAYVILLE",
    "pop" : 45
  }
}
```

6.3.2 Aggregation with User Preference Data

Data Model

Consider a hypothetical sports club with a database that contains a `user` collection that tracks the user's join dates, sport preferences, and stores these data in documents that resemble the following:

```
{
  _id : "jane",
  joined : ISODate("2011-03-02"),
  likes : ["golf", "racquetball"]
}
{
  _id : "joe",
  joined : ISODate("2012-07-02"),
  likes : ["tennis", "golf", "swimming"]
}
```

Normalize and Sort Documents

The following operation returns user names in upper case and in alphabetical order. The aggregation includes user names for all documents in the `users` collection. You might do this to normalize user names for processing.

```
db.users.aggregate(
  [
    { $project : { name:{$toUpper:"$_id"} , _id:0 } },
    { $sort : { name : 1 } }
  ]
)
```

All documents from the `users` collection pass through the pipeline, which consists of the following operations:

- The `$project` operator:
 - creates a new field called `name`.
 - converts the value of the `_id` to upper case, with the `$toUpper` operator. Then the `$project` creates a new field, named `name` to hold this value.
 - suppresses the `id` field. `$project` will pass the `_id` field by default, unless explicitly suppressed.
- The `$sort` operator orders the results by the `name` field.

The results of the aggregation would resemble the following:

```
{
  "name" : "JANE"
},
{
  "name" : "JILL"
},
{
  "name" : "JOE"
}
```

Return Usernames Ordered by Join Month

The following aggregation operation returns user names sorted by the month they joined. This kind of aggregation could help generate membership renewal notices.

```
db.users.aggregate(
  [
    { $project : { month_joined : {
        $month : "$joined"
      },
      name : "$_id",
      _id : 0
    },
    { $sort : { month_joined : 1 } }
  ]
)
```

The pipeline passes all documents in the `users` collection through the following operations:

- The `$project` operator:
 - Creates two new fields: `month_joined` and `name`.
 - Suppresses the `id` from the results. The `aggregate()` method includes the `_id`, unless explicitly suppressed.
- The `$month` operator converts the values of the `joined` field to integer representations of the month. Then the `$project` operator assigns those values to the `month_joined` field.
- The `$sort` operator sorts the results by the `month_joined` field.

The operation returns results that resemble the following:

```
{
  "month_joined" : 1,
  "name" : "ruth"
},
{
  "month_joined" : 1,
```

```
"name" : "harold"
},
{
  "month_joined" : 1,
  "name" : "kate"
}
{
  "month_joined" : 2,
  "name" : "jill"
}
```

Return Total Number of Joins per Month

The following operation shows how many people joined each month of the year. You might use this aggregated data for recruiting and marketing strategies.

```
db.users.aggregate(
  [
    { $project : { month_joined : { $month : "$joined" } } },
    { $group : { _id : {month_joined:"$month_joined"} , number : { $sum : 1 } } },
    { $sort : { "_id.month_joined" : 1 } }
  ]
)
```

The pipeline passes all documents in the `users` collection through the following operations:

- The `$project` operator creates a new field called `month_joined`.
- The `$month` operator converts the values of the `joined` field to integer representations of the month. Then the `$project` operator assigns the values to the `month_joined` field.
- The `$group` operator collects all documents with a given `month_joined` value and counts how many documents there are for that value. Specifically, for each unique value, `$group` creates a new “per-month” document with two fields:
 - `_id`, which contains a nested document with the `month_joined` field and its value.
 - `number`, which is a generated field. The `$sum` operator increments this field by 1 for every document containing the given `month_joined` value.
- The `$sort` operator sorts the documents created by `$group` according to the contents of the `month_joined` field.

The result of this aggregation operation would resemble the following:

```
{
  "_id" : {
    "month_joined" : 1
  },
  "number" : 3
},
{
  "_id" : {
    "month_joined" : 2
  },
  "number" : 9
},
{
  "_id" : {
```

```

    "month_joined" : 3
},
"number" : 5
}
}

```

Return the Five Most Common “Likes”

The following aggregation collects top five most “liked” activities in the data set. This type of analysis could help inform planning and future development.

```

db.users.aggregate(
[
  { $unwind : "$likes" },
  { $group : { _id : "$likes" , number : { $sum : 1 } } },
  { $sort : { number : -1 } },
  { $limit : 5 }
]
)

```

The pipeline begins with all documents in the `users` collection, and passes these documents through the following operations:

- The `$unwind` operator separates each value in the `likes` array, and creates a new version of the source document for every element in the array.

Example

Given the following document from the `users` collection:

```
{
  _id : "jane",
  joined : ISODate("2011-03-02"),
  likes : ["golf", "racquetball"]
}
```

The `$unwind` operator would create the following documents:

```
{
  _id : "jane",
  joined : ISODate("2011-03-02"),
  likes : "golf"
}
{
  _id : "jane",
  joined : ISODate("2011-03-02"),
  likes : "racquetball"
}
```

- The `$group` operator collects all documents the same value for the `likes` field and counts each grouping. With this information, `$group` creates a new document with two fields:
 - `_id`, which contains the `likes` value.
 - `number`, which is a generated field. The `$sum` operator increments this field by 1 for every document containing the given `likes` value.
- The `$sort` operator sorts these documents by the `number` field in reverse order.
- The `$limit` operator only includes the first 5 result documents.

The results of aggregation would resemble the following:

```
{  
  "_id" : "golf",  
  "number" : 33  
,  
{  
  "_id" : "racquetball",  
  "number" : 31  
,  
{  
  "_id" : "swimming",  
  "number" : 24  
,  
{  
  "_id" : "handball",  
  "number" : 19  
,  
{  
  "_id" : "tennis",  
  "number" : 18  
}
```

6.3.3 Map-Reduce Examples

In the mongo shell, the `db.collection.mapReduce()` method is a wrapper around the `mapReduce` command. The following examples use the `db.collection.mapReduce()` method:

Consider the following map-reduce operations on a collection `orders` that contains documents of the following prototype:

```
{  
  _id: ObjectId("50a8240b927d5d8b5891743c"),  
  cust_id: "abc123",  
  ord_date: new Date("Oct 04, 2012"),  
  status: 'A',  
  price: 25,  
  items: [ { sku: "mmm", qty: 5, price: 2.5 },  
           { sku: "nnn", qty: 5, price: 2.5 } ]  
}
```

Return the Total Price Per Customer

Perform the map-reduce operation on the `orders` collection to group by the `cust_id`, and calculate the sum of the `price` for each `cust_id`:

1. Define the map function to process each input document:

- In the function, `this` refers to the document that the map-reduce operation is processing.
- The function maps the `price` to the `cust_id` for each document and emits the `cust_id` and `price` pair.

```
var mapFunction1 = function() {  
  emit(this.cust_id, this.price);  
};
```

2. Define the corresponding reduce function with two arguments `keyCustId` and `valuesPrices`:
 - The `valuesPrices` is an array whose elements are the price values emitted by the map function and grouped by `keyCustId`.
 - The function reduces the `valuesPrice` array to the sum of its elements.

```
var reduceFunction1 = function(keyCustId, valuesPrices) {
    return Array.sum(valuesPrices);
};
```

3. Perform the map-reduce on all documents in the `orders` collection using the `mapFunction1` map function and the `reduceFunction1` reduce function.

```
db.orders.mapReduce(
    mapFunction1,
    reduceFunction1,
    { out: "map_reduce_example" }
)
```

This operation outputs the results to a collection named `map_reduce_example`. If the `map_reduce_example` collection already exists, the operation will replace the contents with the results of this map-reduce operation:

Calculate Order and Total Quantity with Average Quantity Per Item

In this example, you will perform a map-reduce operation on the `orders` collection for all documents that have an `ord_date` value greater than 01/01/2012. The operation groups by the `item.sku` field, and calculates the number of orders and the total quantity ordered for each `sku`. The operation concludes by calculating the average quantity per order for each `sku` value:

1. Define the map function to process each input document:

- In the function, `this` refers to the document that the map-reduce operation is processing.
- For each item, the function associates the `sku` with a new object `value` that contains the `count` of 1 and the item `qty` for the order and emits the `sku` and `value` pair.

```
var mapFunction2 = function() {
    for (var idx = 0; idx < this.items.length; idx++) {
        var key = this.items[idx].sku;
        var value = {
            count: 1,
            qty: this.items[idx].qty
        };
        emit(key, value);
    }
};
```

2. Define the corresponding reduce function with two arguments `keySKU` and `countObjVals`:

- `countObjVals` is an array whose elements are the objects mapped to the grouped `keySKU` values passed by map function to the reducer function.
- The function reduces the `countObjVals` array to a single object `reducedValue` that contains the `count` and the `qty` fields.
- In `reducedVal`, the `count` field contains the sum of the `count` fields from the individual array elements, and the `qty` field contains the sum of the `qty` fields from the individual array elements.

```
var reduceFunction2 = function(keySKU, countObjVals) {
    reducedVal = { count: 0, qty: 0 };

    for (var idx = 0; idx < countObjVals.length; idx++) {
        reducedVal.count += countObjVals[idx].count;
        reducedVal.qty += countObjVals[idx].qty;
    }

    return reducedVal;
};
```

3. Define a finalize function with two arguments key and reducedVal. The function modifies the reducedVal object to add a computed field named avg and returns the modified object:

```
var finalizeFunction2 = function (key, reducedVal) {

    reducedVal.avg = reducedVal.qty/reducedVal.count;

    return reducedVal;

};
```

4. Perform the map-reduce operation on the orders collection using the mapFunction2, reduceFunction2, and finalizeFunction2 functions.

```
db.orders.mapReduce( mapFunction2,
                     reduceFunction2,
                     {
                        out: { merge: "map_reduce_example" },
                        query: { ord_date:
                                { $gt: new Date('01/01/2012') }
                                },
                        finalize: finalizeFunction2
                     }
)
```

This operation uses the query field to select only those documents with ord_date greater than new Date('01/01/2012'). Then it output the results to a collection map_reduce_example. If the map_reduce_example collection already exists, the operation will merge the existing contents with the results of this map-reduce operation.

6.3.4 Perform Incremental Map-Reduce

Map-reduce operations can handle complex aggregation tasks. To perform map-reduce operations, MongoDB provides the mapReduce command and, in the mongo shell, the db.collection.mapReduce() wrapper method.

If the map-reduce data set is constantly growing, you may want to perform an incremental map-reduce rather than performing the map-reduce operation over the entire data set each time.

To perform incremental map-reduce:

1. Run a map-reduce job over the current collection and output the result to a separate collection.
2. When you have more data to process, run subsequent map-reduce job with:
 - the query parameter that specifies conditions that match *only* the new documents.
 - the out parameter that specifies the reduce action to merge the new results into the existing output collection.

Consider the following example where you schedule a map-reduce operation on a `sessions` collection to run at the end of each day.

Data Setup

The `sessions` collection contains documents that log users' sessions each day, for example:

```
db.sessions.save( { userid: "a", ts: ISODate('2011-11-03 14:17:00'), length: 95 } );
db.sessions.save( { userid: "b", ts: ISODate('2011-11-03 14:23:00'), length: 110 } );
db.sessions.save( { userid: "c", ts: ISODate('2011-11-03 15:02:00'), length: 120 } );
db.sessions.save( { userid: "d", ts: ISODate('2011-11-03 16:45:00'), length: 45 } );

db.sessions.save( { userid: "a", ts: ISODate('2011-11-04 11:05:00'), length: 105 } );
db.sessions.save( { userid: "b", ts: ISODate('2011-11-04 13:14:00'), length: 120 } );
db.sessions.save( { userid: "c", ts: ISODate('2011-11-04 17:00:00'), length: 130 } );
db.sessions.save( { userid: "d", ts: ISODate('2011-11-04 15:37:00'), length: 65 } );
```

Initial Map-Reduce of Current Collection

Run the first map-reduce operation as follows:

1. Define the map function that maps the `userid` to an object that contains the fields `userid`, `total_time`, `count`, and `avg_time`:

```
var mapFunction = function() {
    var key = this.userid;
    var value = {
        userid: this.userid,
        total_time: this.length,
        count: 1,
        avg_time: 0
    };

    emit( key, value );
};
```

2. Define the corresponding reduce function with two arguments `key` and `values` to calculate the total time and the count. The `key` corresponds to the `userid`, and the `values` is an array whose elements corresponds to the individual objects mapped to the `userid` in the `mapFunction`.

```
var reduceFunction = function(key, values) {

    var reducedObject = {
        userid: key,
        total_time: 0,
        count: 0,
        avg_time: 0
    };

    values.forEach( function(value) {
        reducedObject.total_time += value.total_time;
        reducedObject.count += value.count;
    });
    return reducedObject;
};
```

3. Define the finalize function with two arguments `key` and `reducedValue`. The function modifies the `reducedValue` document to add another field `average` and returns the modified document.

```
var finalizeFunction = function (key, reducedValue) {  
  
    if (reducedValue.count > 0)  
        reducedValue.avg_time = reducedValue.total_time / reducedValue.count;  
  
    return reducedValue;  
};
```

4. Perform map-reduce on the `sessions` collection using the `mapFunction`, the `reduceFunction`, and the `finalizeFunction` functions. Output the results to a collection `session_stat`. If the `session_stat` collection already exists, the operation will replace the contents:

```
db.sessions.mapReduce( mapFunction,  
                      reduceFunction,  
                      {  
                          out: { reduce: "session_stat" },  
                          finalize: finalizeFunction  
                      }  
)
```

Subsequent Incremental Map-Reduce

Later, as the `sessions` collection grows, you can run additional map-reduce operations. For example, add new documents to the `sessions` collection:

```
db.sessions.save( { userid: "a", ts: ISODate('2011-11-05 14:17:00'), length: 100 } );  
db.sessions.save( { userid: "b", ts: ISODate('2011-11-05 14:23:00'), length: 115 } );  
db.sessions.save( { userid: "c", ts: ISODate('2011-11-05 15:02:00'), length: 125 } );  
db.sessions.save( { userid: "d", ts: ISODate('2011-11-05 16:45:00'), length: 55 } );
```

At the end of the day, perform incremental map-reduce on the `sessions` collection, but use the `query` field to select only the new documents. Output the results to the collection `session_stat`, but reduce the contents with the results of the incremental map-reduce:

```
db.sessions.mapReduce( mapFunction,  
                      reduceFunction,  
                      {  
                          query: { ts: { $gt: ISODate('2011-11-05 00:00:00') } },  
                          out: { reduce: "session_stat" },  
                          finalize: finalizeFunction  
                      }  
);
```

6.3.5 Troubleshoot the Map Function

The `map` function is a JavaScript function that associates or “maps” a value with a key and emits the key and value pair during a [map-reduce](#) (page 280) operation.

To verify the key and value pairs emitted by the `map` function, write your own `emit` function.

Consider a collection `orders` that contains documents of the following prototype:

```
{  
    _id: ObjectId("50a8240b927d5d8b5891743c"),
```

```

    cust_id: "abc123",
    ord_date: new Date("Oct 04, 2012"),
    status: 'A',
    price: 250,
    items: [ { sku: "mmm", qty: 5, price: 2.5 },
              { sku: "nnn", qty: 5, price: 2.5 } ]
}

```

1. Define the `map` function that maps the `price` to the `cust_id` for each document and emits the `cust_id` and `price` pair:

```

var map = function() {
  emit(this.cust_id, this.price);
};

```

2. Define the `emit` function to print the key and value:

```

var emit = function(key, value) {
  print("emit");
  print("key: " + key + " value: " + toJson(value));
}

```

3. Invoke the `map` function with a single document from the `orders` collection:

```

var myDoc = db.orders.findOne( { _id: ObjectId("50a8240b927d5d8b5891743c") } );
map.apply(myDoc);

```

4. Verify the key and value pair is as you expected.

```

emit
key: abc123 value:250

```

5. Invoke the `map` function with multiple documents from the `orders` collection:

```

var myCursor = db.orders.find( { cust_id: "abc123" } );

while (myCursor.hasNext()) {
  var doc = myCursor.next();
  print ("document _id= " + toJson(doc._id));
  map.apply(doc);
  print();
}

```

6. Verify the key and value pairs are as you expected.

See also:

The `map` function must meet various requirements. For a list of all the requirements for the `map` function, see `mapReduce`, or the mongo shell helper method `db.collection.mapReduce()`.

6.3.6 Troubleshoot the Reduce Function

The `reduce` function is a JavaScript function that “reduces” to a single object all the values associated with a particular key during a [map-reduce](#) (page 280) operation. The `reduce` function must meet various requirements. This tutorial helps verify that the `reduce` function meets the following criteria:

- The `reduce` function must return an object whose `type` must be **identical** to the type of the `value` emitted by the `map` function.
- The order of the elements in the `valuesArray` should not affect the output of the `reduce` function.

- The `reduce` function must be *idempotent*.

For a list of all the requirements for the `reduce` function, see `mapReduce`, or the mongo shell helper method `db.collection.mapReduce()`.

Confirm Output Type

You can test that the `reduce` function returns a value that is the same type as the value emitted from the `map` function.

1. Define a `reduceFunction1` function that takes the arguments `keyCustId` and `valuesPrices`. `valuesPrices` is an array of integers:

```
var reduceFunction1 = function(keyCustId, valuesPrices) {
    return Array.sum(valuesPrices);
};
```

2. Define a sample array of integers:

```
var myTestValues = [ 5, 5, 10 ];
```

3. Invoke the `reduceFunction1` with `myTestValues`:

```
reduceFunction1('myKey', myTestValues);
```

4. Verify the `reduceFunction1` returned an integer:

```
20
```

5. Define a `reduceFunction2` function that takes the arguments `keySKU` and `valuesCountObjects`. `valuesCountObjects` is an array of documents that contain two fields `count` and `qty`:

```
var reduceFunction2 = function(keySKU, valuesCountObjects) {
    reducedValue = { count: 0, qty: 0 };

    for (var idx = 0; idx < valuesCountObjects.length; idx++) {
        reducedValue.count += valuesCountObjects[idx].count;
        reducedValue.qty += valuesCountObjects[idx].qty;
    }

    return reducedValue;
};
```

6. Define a sample array of documents:

```
var myTestObjects = [
    { count: 1, qty: 5 },
    { count: 2, qty: 10 },
    { count: 3, qty: 15 }
];
```

7. Invoke the `reduceFunction2` with `myTestObjects`:

```
reduceFunction2('myKey', myTestObjects);
```

8. Verify the `reduceFunction2` returned a document with exactly the `count` and the `qty` field:

```
{ "count" : 6, "qty" : 30 }
```

Ensure Insensitivity to the Order of Mapped Values

The `reduce` function takes a key and a `values` array as its argument. You can test that the result of the `reduce` function does not depend on the order of the elements in the `values` array.

1. Define a sample `values1` array and a sample `values2` array that only differ in the order of the array elements:

```
var values1 = [
    { count: 1, qty: 5 },
    { count: 2, qty: 10 },
    { count: 3, qty: 15 }
];

var values2 = [
    { count: 3, qty: 15 },
    { count: 1, qty: 5 },
    { count: 2, qty: 10 }
];
```

2. Define a `reduceFunction2` function that takes the arguments `keySKU` and `valuesCountObjects`. `valuesCountObjects` is an array of documents that contain two fields `count` and `qty`:

```
var reduceFunction2 = function(keySKU, valuesCountObjects) {
    reducedValue = { count: 0, qty: 0 };

    for (var idx = 0; idx < valuesCountObjects.length; idx++) {
        reducedValue.count += valuesCountObjects[idx].count;
        reducedValue.qty += valuesCountObjects[idx].qty;
    }

    return reducedValue;
};
```

3. Invoke the `reduceFunction2` first with `values1` and then with `values2`:

```
reduceFunction2('myKey', values1);
reduceFunction2('myKey', values2);
```

4. Verify the `reduceFunction2` returned the same result:

```
{ "count" : 6, "qty" : 30 }
```

Ensure Reduce Function Idempotence

Because the map-reduce operation may call a `reduce` multiple times for the same key, and won't call a `reduce` for single instances of a key in the working set, the `reduce` function must return a value of the same type as the value emitted from the `map` function. You can test that the `reduce` function process “reduced” values without affecting the *final* value.

1. Define a `reduceFunction2` function that takes the arguments `keySKU` and `valuesCountObjects`. `valuesCountObjects` is an array of documents that contain two fields `count` and `qty`:

```
var reduceFunction2 = function(keySKU, valuesCountObjects) {
    reducedValue = { count: 0, qty: 0 };

    for (var idx = 0; idx < valuesCountObjects.length; idx++) {
        reducedValue.count += valuesCountObjects[idx].count;
        reducedValue.qty += valuesCountObjects[idx].qty;
    }

}
```

```
        return reducedValue;
    };
}
```

2. Define a sample key:

```
var myKey = 'myKey';
```

3. Define a sample valuesIdempotent array that contains an element that is a call to the reduceFunction2 function:

```
var valuesIdempotent = [
    { count: 1, qty: 5 },
    { count: 2, qty: 10 },
    reduceFunction2(myKey, [ { count: 3, qty: 15 } ] )
];
```

4. Define a sample values1 array that combines the values passed to reduceFunction2:

```
var values1 = [
    { count: 1, qty: 5 },
    { count: 2, qty: 10 },
    { count: 3, qty: 15 }
];
```

5. Invoke the reduceFunction2 first with myKey and valuesIdempotent and then with myKey and values1:

```
reduceFunction2(myKey, valuesIdempotent);
reduceFunction2(myKey, values1);
```

6. Verify the reduceFunction2 returned the same result:

```
{ "count" : 6, "qty" : 30 }
```

6.4 Aggregation Reference

[Aggregation Commands Comparison \(page 304\)](#) A comparison of group, mapReduce and aggregate that explores the strengths and limitations of each aggregation modality.

<http://docs.mongodb.org/manual/reference/operator/aggregation> Aggregation pipeline operations have a collection of operators available to define and manipulate documents in pipeline stages.

[SQL to Aggregation Mapping Chart \(page 306\)](#) An overview common aggregation operations in SQL and MongoDB using the aggregation pipeline and operators in MongoDB and common SQL statements.

[Aggregation Interfaces \(page 308\)](#) The data aggregation interfaces document the invocation format and output for MongoDB's aggregation commands and methods.

6.4.1 Aggregation Commands Comparison

The following table provides a brief overview of the features of the MongoDB aggregation commands.

	aggregate	mapReduce	group
Description	New in version 2.2. Designed with specific goals of improving performance and usability for aggregation tasks. Uses a “pipeline” approach where objects are transformed as they pass through a series of pipeline operators such as \$group, \$match, and \$sort. See Aggregation Reference (page 304) for more information on the pipeline operators.	Implements the Map-Reduce aggregation for processing large data sets.	Provides grouping functionality. Is slower than the aggregate command and has less functionality than the mapReduce command.
Key Features	Pipeline operators can be repeated as needed. Pipeline operators need not produce one output document for every input document. Can also generate new documents or filter out documents.	In addition to grouping operations, can perform complex aggregation tasks as well as perform incremental aggregation on continuously growing datasets. See Map-Reduce Examples (page 296) and Perform Incremental Map-Reduce (page 298).	Can either group by existing fields or with a custom key f JavaScript function, can group by calculated fields. See group for information and example using the key f function.
Flexibility	Limited to the operators and expressions supported by the aggregation pipeline. However, can add computed fields, create new virtual sub-objects, and extract sub-fields into the top-level of results by using the \$project pipeline operator. See \$project for more information as well as Aggregation Reference (page 304) for more information on all the available pipeline operators.	Custom map, reduce and finalize JavaScript functions offer flexibility to aggregation logic. See mapReduce for details and restrictions on the functions.	Custom reduce and finalize JavaScript functions offer flexibility to grouping logic. See group for details and restrictions on these functions.
Output Results	Returns results inline. The result is subject to the <i>BSON Document size</i> limit.	Returns results in various options (inline, new collection, merge, replace, reduce). See mapReduce for details on the output options. Changed in version 2.2: Provides much better support for sharded map-reduce output than previous versions.	Returns results inline as an array of grouped items. The result set must fit within the <i>maximum BSON document size limit</i> . Changed in version 2.2: The returned array can contain at most 20,000 elements; i.e. at most 20,000 unique groupings. Previous versions had a limit of 10,000 elements.
Sharding Notes	Supports non-sharded and sharded input collections.	Supports non-sharded and sharded input collections. Prior to 2.4, JavaScript code executed in a single thread.	Does not support sharded collection. Prior to 2.4, JavaScript code executed in a single thread.
More Information	See Aggregation Concepts (page 277) and aggregate.	See Map-Reduce (page 280) and mapReduce.	See group.
6.4a-Aggregation Reference			305

6.4.2 SQL to Aggregation Mapping Chart

The [aggregation pipeline](#) (page 277) allows MongoDB to provide native aggregation capabilities that corresponds to many common data aggregation operations in SQL. If you're new to MongoDB you might want to consider the [Frequently Asked Questions](#) (page 571) section for a selection of common questions.

The following table provides an overview of common SQL aggregation terms, functions, and concepts and the corresponding MongoDB *aggregation operators*:

SQL Terms, Functions, and Concepts	MongoDB Aggregation Operators
WHERE	\$match
GROUP BY	\$group
HAVING	\$match
SELECT	\$project
ORDER BY	\$sort
LIMIT	\$limit
SUM()	\$sum
COUNT()	\$sum
join	No direct corresponding operator; <i>however</i> , the \$unwind operator allows for somewhat similar functionality, but with fields embedded within the document.

Examples

The following table presents a quick reference of SQL aggregation statements and the corresponding MongoDB statements. The examples in the table assume the following conditions:

- The SQL examples assume *two* tables, `orders` and `order_lineitem` that join by the `order_lineitem.order_id` and the `orders.id` columns.
- The MongoDB examples assume *one* collection `orders` that contain documents of the following prototype:

```
{
  cust_id: "abc123",
  ord_date: ISODate("2012-11-02T17:04:11.102Z"),
  status: 'A',
  price: 50,
  items: [ { sku: "xxx", qty: 25, price: 1 },
            { sku: "yyy", qty: 25, price: 1 } ]
}
```

- The MongoDB statements prefix the names of the fields from the *documents* in the collection `orders` with a `$` character when they appear as operands to the aggregation operations.

SQL Example	MongoDB Example	Description
<pre>SELECT COUNT(*) AS count FROM orders</pre>	<pre>db.orders.aggregate([{ \$group: { _id: null, count: { \$sum: 1 } } }]</pre>	Count all records from orders
<pre>SELECT SUM(price) AS total FROM orders</pre>	<pre>db.orders.aggregate([{ \$group: { _id: null, total: { \$sum: "\$price" } } }]</pre>	Sum the price field from orders
<pre>SELECT cust_id, SUM(price) AS total FROM orders GROUP BY cust_id</pre>	<pre>db.orders.aggregate([{ \$group: { _id: "\$cust_id", total: { \$sum: "\$price" } } }]</pre>	For each unique cust_id, sum the price field.
<pre>SELECT cust_id, SUM(price) AS total FROM orders GROUP BY cust_id ORDER BY total</pre>	<pre>db.orders.aggregate([{ \$group: { _id: "\$cust_id", total: { \$sum: "\$price" } }, { \$sort: { total: 1 } }]</pre>	For each unique cust_id, sum the price field, results sorted by sum.
<pre>SELECT cust_id, ord_date, SUM(price) AS total FROM orders GROUP BY cust_id, ord_date</pre>	<pre>db.orders.aggregate([{ \$group: { _id: { cust_id: "\$cust_id", ord_date: "\$ord_date" }, total: { \$sum: "\$price" } } }]</pre>	For each unique cust_id, ord_date grouping, sum the price field.
<pre>SELECT cust_id, count(*) FROM orders GROUP BY cust_id HAVING count(*) > 1</pre>	<pre>db.orders.aggregate([{ \$group: { _id: "\$cust_id", count: { \$sum: 1 } }, { \$match: { count: { \$gt: 1 } } }]</pre>	For cust_id with multiple records, return the cust_id and the corresponding record count.
<pre>SELECT cust_id, ord_date, SUM(price) AS total FROM orders GROUP BY cust_id, ord_date HAVING total > 250</pre>	<pre>db.orders.aggregate([{ \$group: { _id: { cust_id: "\$cust_id", ord_date: "\$ord_date" }, total: { \$sum: "\$price" } }, { \$match: { total: { \$gt: 250 } } }]</pre>	For each unique cust_id, ord_date grouping, sum the price field and return only where the sum is greater than 250.
<pre>SELECT cust_id, SUM(price) AS total FROM orders WHERE status = 'A' GROUP BY cust_id</pre>	<pre>db.orders.aggregate([{ \$match: { status: 'A' } }, { \$group: { _id: "\$cust_id", total: { \$sum: "\$price" } } }]</pre>	For each unique cust_id with status A, sum the price field.
<pre>SELECT cust_id, SUM(price) AS total FROM orders WHERE status = 'A'</pre>	<pre>db.orders.aggregate([{ \$match: { status: 'A' } }, { \$group: { _id: "\$cust_id", total: { \$sum: "\$price" } } }, { \$match: { total: { \$gt: 250 } } }]</pre>	For each unique cust_id with status A, sum the price field and return only where the sum is greater than 250.
6.4 GROUP BY Reference		
<pre>HAVING total > 250</pre>		
<pre>SELECT cust_id,</pre>	<pre>db.orders.aggregate([</pre>	For each unique cust_id, sum the corresponding line item price fields.

6.4.3 Aggregation Interfaces

Aggregation Commands

Name	Description
aggregate	Performs aggregation tasks (page 277) such as group using the aggregation framework.
count	Counts the number of documents in a collection.
distinct	Displays the distinct values found for a specified key in a collection.
group	Groups documents in a collection by the specified key and performs simple aggregation.
mapReduce	Performs map-reduce (page 280) aggregation for large data sets.

Aggregation Methods

Name	Description
db.collection.aggregate()	Provides access to the aggregation pipeline (page 277).
db.collection.group()	Groups documents in a collection by the specified key and performs simple aggregation.
db.collection.mapReduce()	Performs map-reduce (page 280) aggregation for large data sets.

Indexes

Indexes provide high performance read operations for frequently used queries.

This section introduces indexes in MongoDB, describes the types and configuration options for indexes, and describes special types of indexing MongoDB supports. The section also provides tutorials detailing procedures and operational concerns, and providing information on how applications may use indexes.

[**Index Introduction** \(page 309\)](#) An introduction to indexes in MongoDB.

[**Index Concepts** \(page 314\)](#) The core documentation of indexes in MongoDB, including geospatial and text indexes.

[**Index Types** \(page 315\)](#) MongoDB provides different types of indexes for different purposes and different types of content.

[**Index Properties** \(page 330\)](#) The properties you can specify when building indexes.

[**Index Creation** \(page 332\)](#) The options available when creating indexes.

[**Indexing Tutorials** \(page 334\)](#) Examples of operations involving indexes, including index creation and querying indexes.

[**Indexing Reference** \(page 370\)](#) Reference material for indexes in MongoDB.

7.1 Index Introduction

Indexes support the efficient resolution of queries in MongoDB. Without indexes, MongoDB must scan every document in a collection to select those documents that match the query statement. These *collection scans* are inefficient and require the mongod to process a large volume of data for each operation.

Indexes are special data structures ¹ that store a small portion of the collection's data set in an easy to traverse form. The index stores the value of a specific field or set of fields, ordered by the value of the field.

Fundamentally, indexes in MongoDB are similar to indexes in other database systems. MongoDB defines indexes at the *collection* level and supports indexes on any field or sub-field of the documents in a MongoDB collection.

If an appropriate index exists for a query, MongoDB can use the index to limit the number of documents it must inspect. In some cases, MongoDB can use the data from the index to determine which documents match a query. The following diagram illustrates a query that selects documents using an index.

Consider the documentation of the *query optimizer* (page 35) for more information on the relationship between queries and indexes.

Tip

¹ MongoDB indexes use a B-tree data structure.

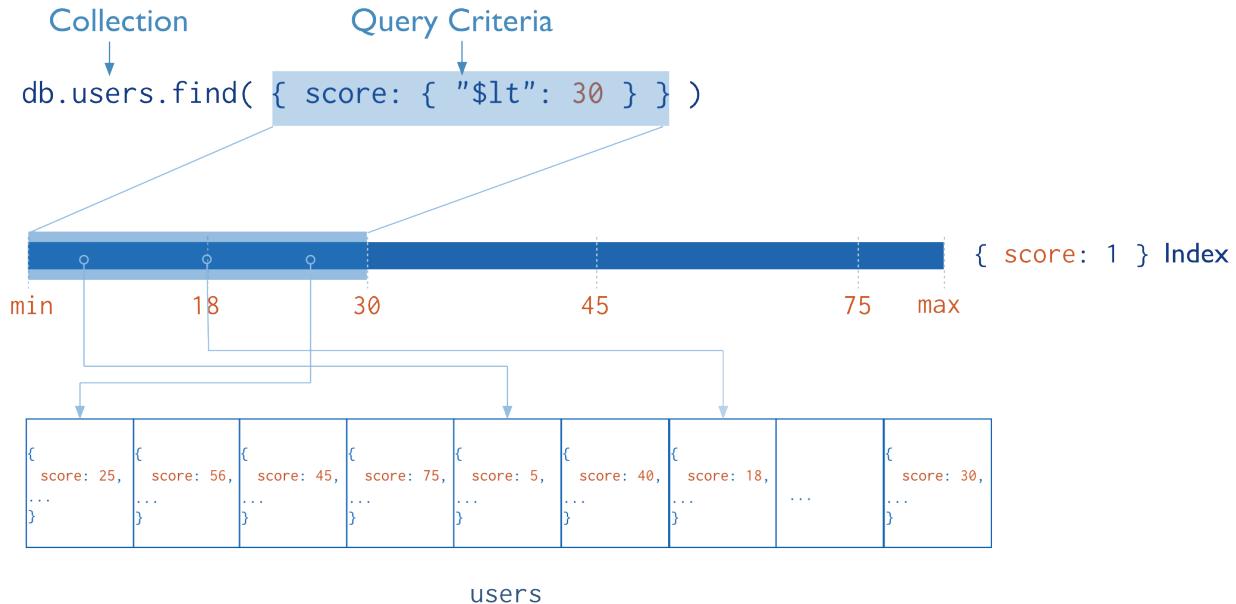


Figure 7.1: Diagram of a query selecting documents using an index. MongoDB narrows the query by scanning the range of documents with values of `score` less than 30.

Create indexes to support common and user-facing queries. Having these indexes will ensure that MongoDB only scans the smallest possible number of documents.

Indexes can also optimize the performance of other operations in specific situations:

Sorted Results

MongoDB can use indexes to return documents sorted by the index key directly from the index without requiring an additional sort phase.

Covered Results

When the query criteria and the *projection* of a query include *only* the indexed fields, MongoDB will return results directly from the index *without* scanning any documents or bringing documents into memory. These covered queries can be *very* efficient. Indexes can also cover *aggregation pipeline operations* (page 277).

7.1.1 Index Types

MongoDB provides a number of different index types to support specific types of data and queries.

Default `_id`

All MongoDB collections have an index on the `_id` field that exists by default. If applications do not specify a value for `_id` the driver or the `mongod` will create an `_id` field with an *ObjectID* value.

The `_id` index is *unique*, and prevents clients from inserting two documents with the same value for the `_id` field.

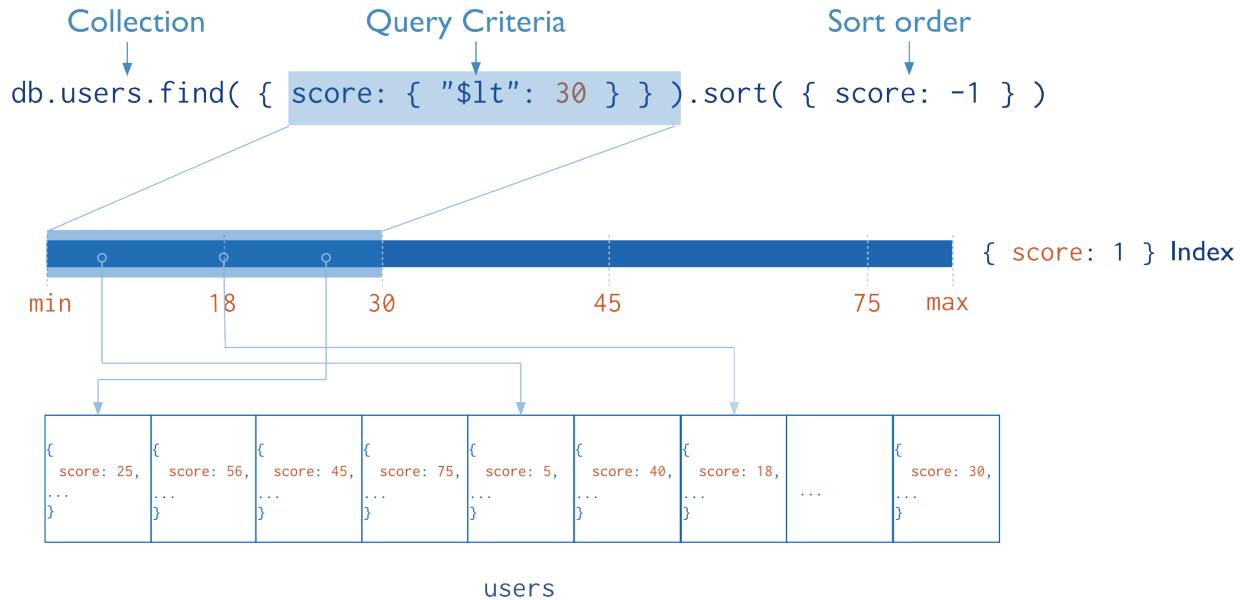


Figure 7.2: Diagram of a query that uses an index to select and return sorted results. The index stores `score` values in ascending order. MongoDB can traverse the index in either ascending or descending order to return sorted results.

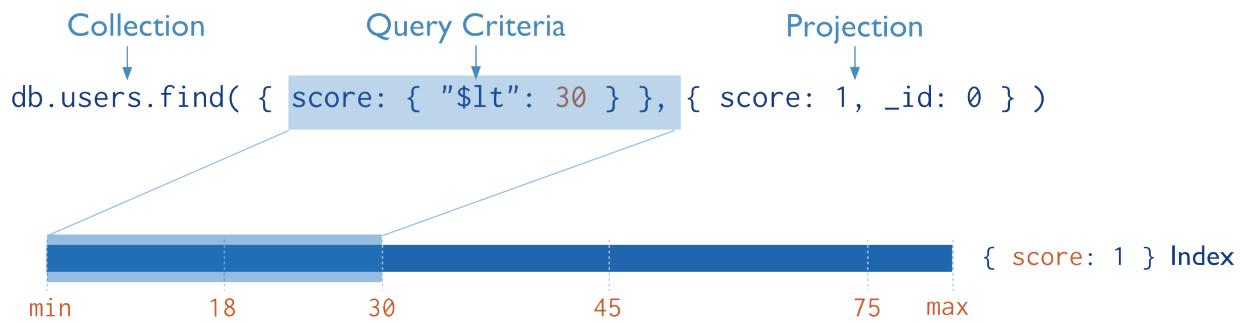


Figure 7.3: Diagram of a query that uses only the index to match the query criteria and return the results. MongoDB does not need to inspect data outside of the index to fulfill the query.

Single Field

In addition to the MongoDB-defined `_id` index, MongoDB supports user-defined indexes on a *single field of a document* (page 316). Consider the following illustration of a single-field index:

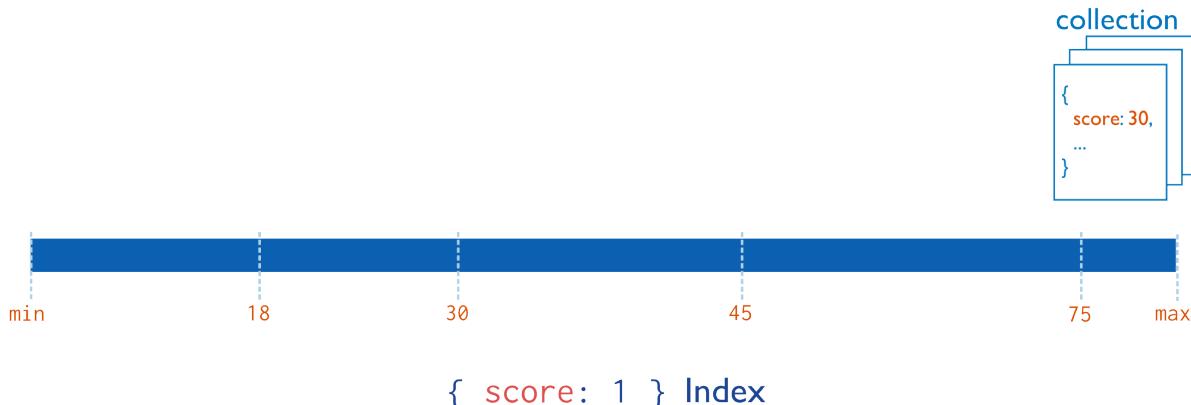


Figure 7.4: Diagram of an index on the `score` field (ascending).

Compound Index

MongoDB also supports user-defined indexes on multiple fields. These *compound indexes* (page 318) behave like single-field indexes; however, the query can select documents based on additional fields. The order of fields listed in a compound index has significance. For instance, if a compound index consists of `{ userid: 1, score: -1 }`, the index sorts first by `userid` and then, within each `userid` value, sort by `score`. Consider the following illustration of this compound index:

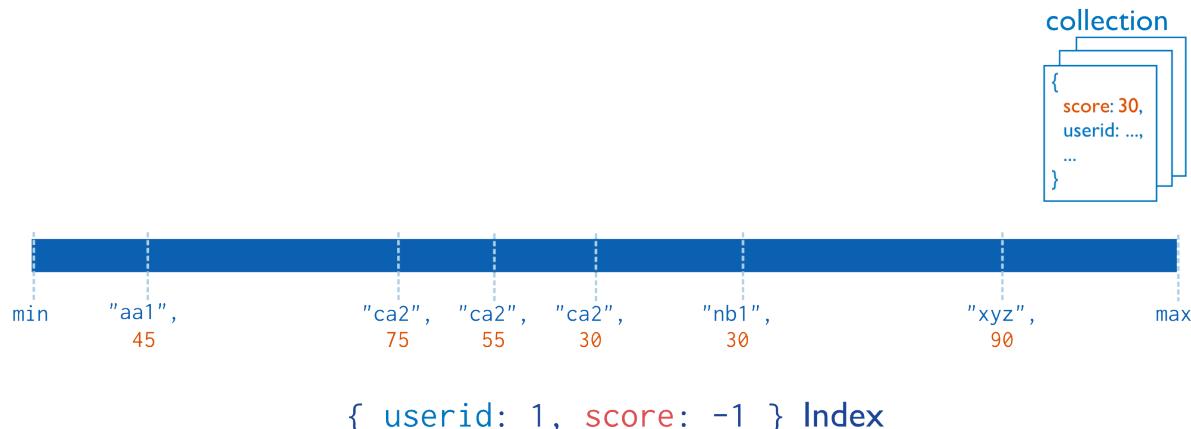


Figure 7.5: Diagram of a compound index on the `userid` field (ascending) and the `score` field (descending). The index sorts first by the `userid` field and then by the `score` field.

Multikey Index

MongoDB uses [multikey indexes](#) (page 320) to index the content stored in arrays. If you index a field that holds an array value, MongoDB creates separate index entries for *every* element of the array. These [multikey indexes](#) (page 320) allow queries to select documents that contain arrays by matching on element or elements of the arrays. MongoDB automatically determines whether to create a multikey index if the indexed field contains an array value; you do not need to explicitly specify the multikey type.

Consider the following illustration of a multikey index:

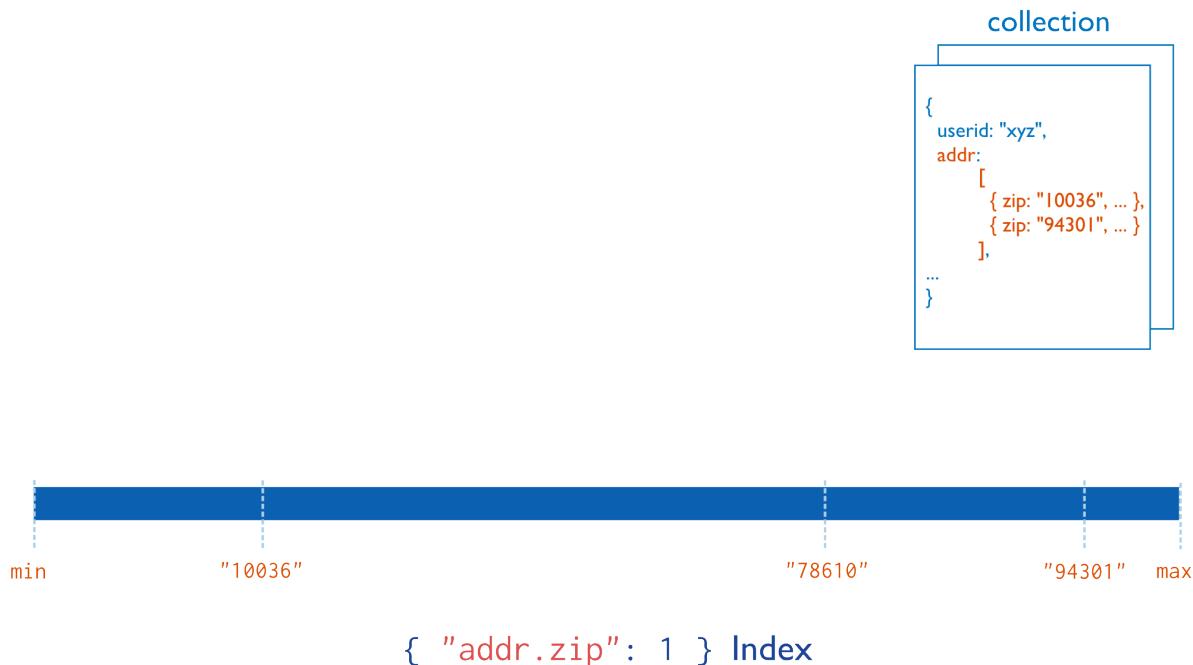


Figure 7.6: Diagram of a multikey index on the `addr.zip` field. The `addr` field contains an array of address documents. The address documents contain the `zip` field.

Geospatial Index

To support efficient queries of geospatial coordinate data, MongoDB provides two special indexes: [2d indexes](#) (page 325) that uses planar geometry when returning results and [2sphere indexes](#) (page 324) that use spherical geometry to return results.

See [2d Index Internals](#) (page 327) for a high level introduction to geospatial indexes.

Text Indexes

MongoDB provides a *beta* `text` index type that supports searching for string content in a collection. These text indexes do not store language-specific *stop* words (e.g. “the”, “a”, “or”) and *stem* the words in a collection to only store root words.

See [Text Indexes](#) (page 328) for more information on text indexes and search.

Hashed Indexes

To support [hash based sharding](#) (page 500), MongoDB provides a [hashed index](#) (page 329) type, which indexes the hash of the value of a field. These indexes have a more random distribution of values along their range, but *only* support equality matches and cannot support range-based queries.

7.1.2 Index Properties

Unique Indexes

The [unique](#) (page 330) property for an index causes MongoDB to reject duplicate values for the indexed field. To create a [unique index](#) (page 330) on a field that already has duplicate values, see [Drop Duplicates](#) (page 334) for index creation options. Other than the unique constraint, unique indexes are functionally interchangeable with other MongoDB indexes.

Sparse Indexes

The [sparse](#) (page 331) property of an index ensures that the index only contain entries for documents that have the indexed field. The index skips documents that *do not* have the indexed field.

You can combine the sparse index option with the unique index option to reject documents that have duplicate values for a field but ignore documents that do not have the indexed key.

7.2 Index Concepts

These documents describe and provide examples of the types, configuration options, and behavior of indexes in MongoDB. For an over view of indexing, see [Index Introduction](#) (page 309). For operational instructions, see [Indexing Tutorials](#) (page 334). The [Indexing Reference](#) (page 370) documents the commands and operations specific to index construction, maintenance, and querying in MongoDB, including index types and creation options.

[Index Types](#) (page 315) MongoDB provides different types of indexes for different purposes and different types of content.

[Single Field Indexes](#) (page 316) A single field index only includes data from a single field of the documents in a collection. MongoDB supports single field indexes on fields at the top level of a document *and* on fields in sub-documents.

[Compound Indexes](#) (page 318) A compound index includes more than one field of the documents in a collection.

[Multikey Indexes](#) (page 320) A multikey index references an array and records a match if a query includes any value in the array.

[Geospatial Indexes and Queries](#) (page 322) Geospatial indexes support location-based searches on data that is stored as either GeoJSON objects or legacy coordinate pairs.

[Text Indexes](#) (page 328) Text indexes supports search of string content in documents.

[Hashed Index](#) (page 329) Hashed indexes maintain entries with hashes of the values of the indexed field.

[Index Properties](#) (page 330) The properties you can specify when building indexes.

[TTL Indexes](#) (page 330) The TTL index is used for TTL collections, which expire data after a period of time.

[Unique Indexes](#) (page 330) A unique index causes MongoDB to reject all documents that contain a duplicate value for the indexed field.

[Sparse Indexes](#) (page 331) A sparse index does not index documents that do not have the indexed field.

[Index Creation](#) (page 332) The options available when creating indexes.

7.2.1 Index Types

MongoDB provides a number of different index types. You can create indexes on any field or embedded field within a document or sub-document. You can create [single field indexes](#) (page 316) or [compound indexes](#) (page 318). MongoDB also supports indexes of arrays, called [multi-key indexes](#) (page 320), as well as supports [indexes on geospatial data](#) (page 322). For a list of the supported index types, see [Index Type Documentation](#) (page 315).

In general, you should create indexes that support your common and user-facing queries. Having these indexes will ensure that MongoDB scans the smallest possible number of documents.

In the mongo shell, you can create an index by calling the `ensureIndex()` method. For more detailed instructions about building indexes, see the [Indexing Tutorials](#) (page 334) page.

Behavior of Index Types

All indexes in MongoDB are *B-tree* indexes, which can efficiently support equality matches and range queries. The index stores items internally in order sorted by the value of the index field. The ordering of index entries supports efficient range-based operations and allows MongoDB to return sorted results using the order of documents in the index.

Ordering of Indexes

MongoDB indexes may be ascending, (i.e. 1) or descending (i.e. -1) in their ordering. Nevertheless, MongoDB may also traverse the index in either directions. As a result, for single-field indexes, ascending and descending indexes are interchangeable. This is not the case for compound indexes: in compound indexes, the direction of the sort order can have a greater impact on the results.

See [Sort Order](#) (page 319) for more information on the impact of index order on results in compound indexes.

Redundant Indexes

A single query can only use *one* index, except for queries that use the `$or` operator that can use a different index for each clause.

See also:

[Index Limitations](#).

Index Type Documentation

[Single Field Indexes](#) (page 316) A single field index only includes data from a single field of the documents in a collection. MongoDB supports single field indexes on fields at the top level of a document *and* on fields in sub-documents.

[Compound Indexes](#) (page 318) A compound index includes more than one field of the documents in a collection.

[Multikey Indexes](#) (page 320) A multikey index references an array and records a match if a query includes any value in the array.

Geospatial Indexes and Queries (page 322) Geospatial indexes support location-based searches on data that is stored as either GeoJSON objects or legacy coordinate pairs.

Text Indexes (page 328) Text indexes supports search of string content in documents.

Hashed Index (page 329) Hashed indexes maintain entries with hashes of the values of the indexed field.

Single Field Indexes

MongoDB provides complete support for indexes on any field in a *collection of documents*. By default, all collections have an index on the [_id field](#) (page 317), and applications and users may add additional indexes to support important queries and operations.

MongoDB supports indexes that contain either a single field *or* multiple fields depending on the operations that index supports. This document describes indexes that contain a single field. Consider the following illustration of a single field index.

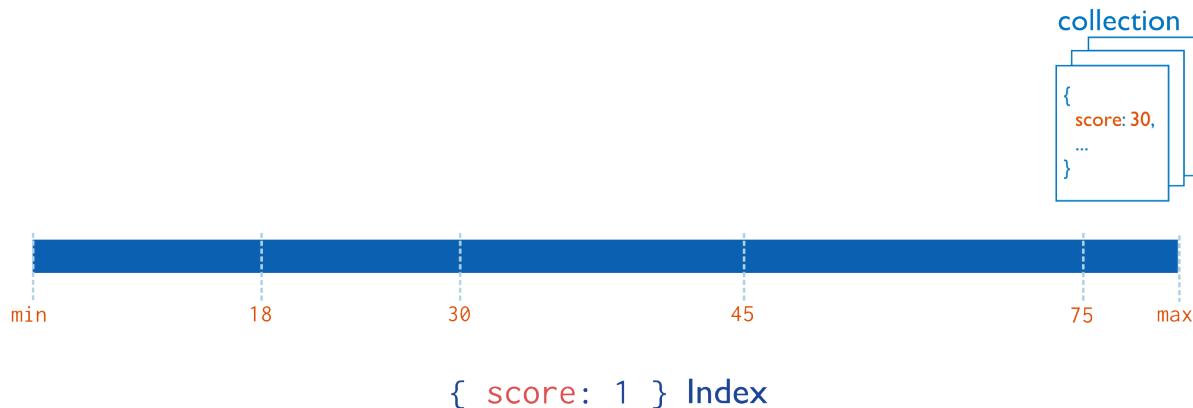


Figure 7.7: Diagram of an index on the `score` field (ascending).

See also:

[Compound Indexes](#) (page 318) for information about indexes that include multiple fields, and [Index Introduction](#) (page 309) for a higher level introduction to indexing in MongoDB.

Example Given the following document in the `friends` collection:

```
{ "_id" : ObjectId(...),
  "name" : "Alice"
  "age" : 27
}
```

The following command creates an index on the `name` field:

```
db.friends.ensureIndex( { "name" : 1 } )
```

Cases

_id Field Index For all collections, MongoDB creates the default `_id` index, which is a *unique index* (page 330) on the `_id` field. MongoDB creates this index by default on all collections. You cannot delete the index on `_id`.

You can think of the `_id` field as the *primary key* for the collection. Every document *must* have a unique `_id` field. You may store any unique value in the `_id` field. The default value of `_id` is an *ObjectId* on every `insert()` operation. An *ObjectId* is a 12-byte unique identifier suitable for use as the value of an `_id` field.

Note: In sharded clusters, if you do *not* use the `_id` field as the *shard key*, then your application **must** ensure the uniqueness of the values in the `_id` field to prevent errors. This is most-often done by using a standard auto-generated *ObjectId*.

Before version 2.2, *capped collections* did not have an `_id` field. In version 2.2 and newer, capped collection do have an `_id` field, except those in the `local` database. See *Capped Collections Recommendations and Restrictions* (page 158) for more information.

Indexes on Embedded Fields You can create indexes on fields embedded in sub-documents, just as you can index top-level fields in documents. Indexes on embedded fields differ from *indexes on sub-documents* (page 317), which include the full content up to the maximum `Index Size` of the sub-document in the index. Instead, indexes on embedded fields allow you to use a “dot notation,” to introspect into sub-documents.

Consider a collection named `people` that holds documents that resemble the following example document:

```
{ "_id": ObjectId(...),
  "name": "John Doe",
  "address": {
    "street": "Main",
    "zipcode": 53511,
    "state": "WI"
  }
}
```

You can create an index on the `address.zipcode` field, using the following specification:

```
db.people.ensureIndex( { "address.zipcode": 1 } )
```

Indexes on Subdocuments You can also create indexes on subdocuments.

For example, the `factories` collection contains documents that contain a `metro` field, such as:

```
{
  _id: ObjectId("523cba3c73a8049bcd6007"),
  metro: {
    city: "New York",
    state: "NY"
  },
  name: "Giant Factory"
}
```

The `metro` field is a subdocument, containing the embedded fields `city` and `state`. The following creates an index on the `metro` field as a whole:

```
db.factories.ensureIndex( { metro: 1 } )
```

The following query can use the index on the `metro` field:

```
db.factories.find( { metro: { city: "New York", state: "NY" } } )
```

This query returns the above document. When performing equality matches on subdocuments, field order matters and the subdocuments must match exactly. For example, the following query does not match the above document:

```
db.factories.find( { metro: { state: "NY", city: "New York" } } )
```

See [query-subdocuments](#) for more information regarding querying on subdocuments.

The index on the `metro` field can also support the following query:

```
db.factories.find( { metro: { $gte : { city: "New York" } } } )
```

This query returns the above document because `{ city: "New York", state: "NY" }` is greater than `{ city: "New York" }`. The order of comparison is in ascending key order in the order that the keys occur in the `BSON` document.

Compound Indexes

MongoDB supports *compound indexes*, where a single index structure holds references to multiple fields² within a collection's documents. The following diagram illustrates an example of a compound index on two fields:

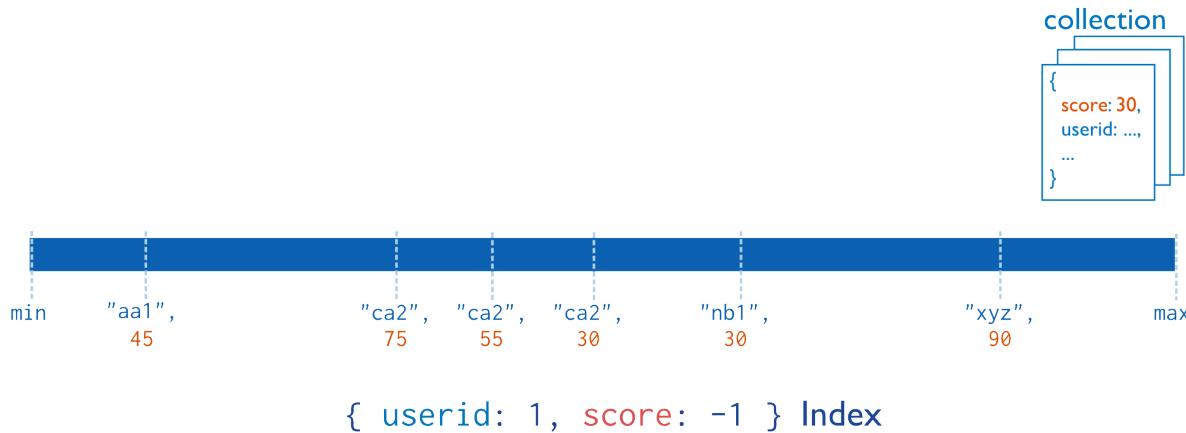


Figure 7.8: Diagram of a compound index on the `userid` field (ascending) and the `score` field (descending). The index sorts first by the `userid` field and then by the `score` field.

Compound indexes can support queries that match on multiple fields.

Example

Consider a collection named `products` that holds documents that resemble the following document:

```
{
  "_id": ObjectId(...),
  "item": "Banana",
  "category": ["food", "produce", "grocery"],
  "location": "4th Street Store",
  "stock": 4,
  "type": cases,
  "arrival": Date(...)
}
```

² MongoDB imposes a limit of 31 fields for any compound index.

If applications query on the `item` field as well as query on both the `item` field and the `stock` field, you can specify a single compound index to support both of these queries:

```
db.products.ensureIndex( { "item": 1, "stock": 1 } )
```

Important: You may not create compound indexes that have hashed index fields. You will receive an error if you attempt to create a compound index that includes [a hashed index](#) (page 329).

The order of the fields in a compound index is very important. In the previous example, the index will contain references to documents sorted first by the values of the `item` field and, within each value of the `item` field, sorted by values of the `stock` field. See [Sort Order](#) (page 319) for more information.

In addition to supporting queries that match on all the index fields, compound indexes can support queries that match on the prefix of the index fields. For details, see [Prefixes](#) (page 319).

Sort Order Indexes store references to fields in either ascending (1) or descending (-1) sort order. For single-field indexes, the sort order of keys doesn't matter because MongoDB can traverse the index in either direction. However, for [compound indexes](#) (page 318), sort order can matter in determining whether the index can support a sort operation.

Consider a collection `events` that contains documents with the fields `username` and `date`. Applications can issue queries that return results sorted first by ascending `username` values and then by descending (i.e. more recent to last) `date` values, such as:

```
db.events.find().sort( { username: 1, date: -1 } )
```

or queries that return results sorted first by descending `username` values and then by ascending `date` values, such as:

```
db.events.find().sort( { username: -1, date: 1 } )
```

The following index can support both these sort operations:

```
db.events.ensureIndex( { "username" : 1, "date" : -1 } )
```

However, the above index cannot support sorting by ascending `username` values and then by ascending `date` values, such as the following:

```
db.events.find().sort( { username: 1, date: 1 } )
```

Prefixes Compound indexes support queries on any prefix of the index fields. Index prefixes are the beginning subset of indexed fields. For example, given the index `{ a: 1, b: 1, c: 1 }`, both `{ a: 1 }` and `{ a: 1, b: 1 }` are prefixes of the index.

If you have a collection that has a compound index on `{ a: 1, b: 1 }`, as well as an index that consists of the prefix of that index, i.e. `{ a: 1 }`, assuming none of the index has a sparse or unique constraints, then you can drop the `{ a: 1 }` index. MongoDB will be able to use the compound index in all of situations that it would have used the `{ a: 1 }` index.

Example

Given the following index:

```
{ "item": 1, "location": 1, "stock": 1 }
```

MongoDB **can** use this index to support queries that include:

- the `item` field, and

- the `item` field *and* the `location` field,
- the `item` field *and* the `location` field *and* the `stock` field, and
- only the `item` *and* `stock` fields; however, this index would be less efficient than an index on only `item` and `stock`.

MongoDB **cannot** use this index to support queries that include:

- only the `location` field,
- only the `stock` field,
- only the `location` *and* `stock` fields, and

Multikey Indexes

To index a field that holds an array value, MongoDB adds index items for each item in the array. These *multikey* indexes allow MongoDB return documents from queries using the value of an array. MongoDB automatically determines whether to create a multikey index if the indexed field contains an array value; you do not need to explicitly specify the multikey type.

Consider the following illustration of a multikey index:

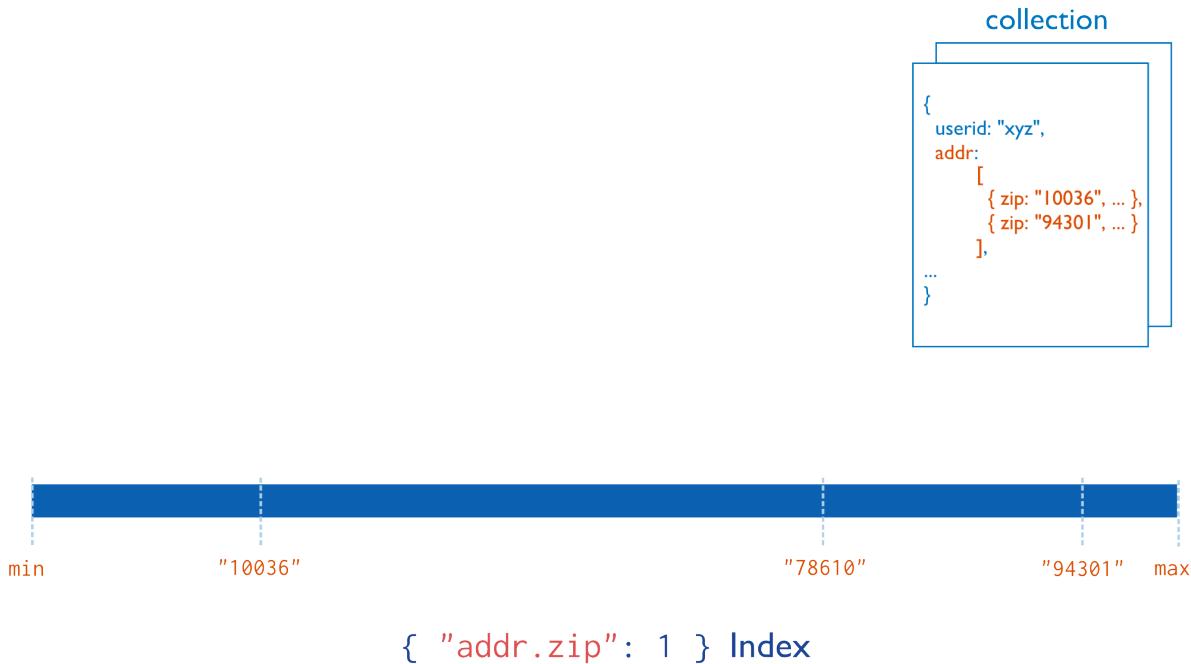


Figure 7.9: Diagram of a multikey index on the `addr.zip` field. The `addr` field contains an array of address documents. The address documents contain the `zip` field.

Multikey indexes support all operations of other MongoDB indexes; however, applications may use multikey indexes to select documents based on ranges of values for the value of an array. Multikey indexes support arrays that hold both values (e.g. strings, numbers) *and* nested documents.

Limitations

Interactions between Compound and Multikey Indexes While you can create multikey *compound indexes* (page 318), at most one field in a compound index may hold an array. For example, given an index on { a: 1, b: [1] }, the following documents are permissible:

```
{a: [1, 2], b: 1}
{a: 1, b: [1, 2]}
```

However, the following document is impermissible, and MongoDB cannot insert such a document into a collection with the {a: 1, b: [1]} index:

```
{a: [1, 2], b: [1, 2]}
```

If you attempt to insert a such a document, MongoDB will reject the insertion, and produce an error that says `cannot index parallel arrays`. MongoDB does not index parallel arrays because they require the index to include each value in the Cartesian product of the compound keys, which could quickly result in incredibly large and difficult to maintain indexes.

Shard Keys

Important: The index of a shard key **cannot** be a multi-key index.

Hashed Indexes hashed indexes are not compatible with multi-key indexes.

To compute the hash for a hashed index, MongoDB collapses sub-documents and computes the hash for the entire value. For fields that hold arrays or sub-documents, you cannot use the index to support queries that introspect the sub-document.

Examples

Index Basic Arrays Given the following document:

```
{
  "_id" : ObjectId("..."),
  "name" : "Warm Weather",
  "author" : "Steve",
  "tags" : [ "weather", "hot", "record", "april" ]
}
```

Then an index on the tags field, { tags: 1 }, would be a multikey index and would include these four separate entries for that document:

- "weather",
- "hot",
- "record", and
- "april".

Queries could use the multikey index to return queries for any of the above values.

Index Arrays with Nested Documents You can use multikey indexes to index fields within objects embedded in arrays, as in the following example:

Consider a feedback collection with documents in the following form:

```
{  
  "_id": ObjectId(...)  
  "title": "Grocery Quality"  
  "comments": [  
    { author_id: ObjectId(...)  
     date: Date(...)  
     text: "Please expand the cheddar selection." },  
    { author_id: ObjectId(...)  
     date: Date(...)  
     text: "Please expand the mustard selection." },  
    { author_id: ObjectId(...)  
     date: Date(...)  
     text: "Please expand the olive selection." }  
  ]  
}
```

An index on the `comments.text` field would be a multikey index and would add items to the index for all of the sub-documents in the array.

With an index, such as `{ comments.text: 1 }`, consider the following query:

```
db.feedback.find( { "comments.text": "Please expand the olive selection." } )
```

This would select the document, that contains the following document in the `comments.text` array:

```
{ author_id: ObjectId(...)  
  date: Date(...)  
  text: "Please expand the olive selection." }
```

Geospatial Indexes and Queries

MongoDB offers a number of indexes and query mechanisms to handle geospatial information. This section introduces MongoDB's geospatial features. For complete examples of geospatial queries in MongoDB, see [Geospatial Index Tutorials](#) (page 345).

Surfaces Before storing your location data and writing queries, you must decide the type of surface to use to perform calculations. The type you choose affects how you store data, what type of index to build, and the syntax of your queries.

MongoDB offers two surface types:

Spherical To calculate geometry over an Earth-like sphere, store your location data on a spherical surface and use [2dsphere](#) (page 324) index.

Store your location data as GeoJSON objects with this coordinate-axis order: **longitude, latitude**. The coordinate reference system for GeoJSON uses the [WGS84](#) datum.

Flat To calculate distances on a Euclidean plane, store your location data as legacy coordinate pairs and use a [2d](#) (page 325) index.

Location Data If you choose spherical surface calculations, you store location data as either:

GeoJSON Objects Queries on *GeoJSON* objects always calculate on a sphere. The default coordinate reference system for GeoJSON uses the *WGS84* datum.

New in version 2.4: Support for GeoJSON storage and queries is new in version 2.4. Prior to version 2.4, all geospatial data used coordinate pairs.

MongoDB supports the following GeoJSON objects:

- Point
- LineString
- Polygon

Legacy Coordinate Pairs MongoDB csupports spherical surface calculations on *legacy coordinate pairs* by converting the data to the GeoJSON Point type.

If you choose flat surface calculations, you can store data only as *legacy coordinate pairs*.

Query Operations MongoDB’s geospatial query operators let you query for:

Inclusion MongoDB can query for locations contained entirely within a specified polygon. Inclusion queries use the `$geoWithin` operator.

Intersection MongoDB can query for locations that intersect with a specified geometry. These queries apply only to data on a spherical surface. These queries use the `$geoIntersects` operator.

Proximity MongoDB can query for the points nearest to another point. Proximity queries use the `$near` operator. The `$near` operator requires a `2d` or `2dsphere` index.

Geospatial Indexes MongoDB provides the following geospatial index types to support the geospatial queries.

2dsphere *2dsphere* (page 324) indexes support:

- Calculations on a sphere
- Both GeoJSON objects and legacy coordinate pairs
- A compound index with scalar index fields (i.e. ascending or descending) as a prefix or suffix of the `2dsphere` index field

New in version 2.4: `2dsphere` indexes are not available before version 2.4.

See also:

Query a 2dsphere Index (page 346)

2d *2d* (page 325) indexes support:

- Calculations using flat geometry
- Legacy coordinate pairs (i.e., geospatial points on a flat coordinate system)
- A compound index with only one additional field, as a suffix of the `2d` index field

See also:

Query a 2d Index (page 349)

Geospatial Indexes and Sharding You *cannot* use a geospatial index as the *shard key* index.

You can create and maintain a geospatial index on a sharded collection if using different fields as the shard key.

Queries using `$near` are not supported for sharded collections. Use `geoNear` instead. You also can query for geospatial data using `$geoWithin`.

Additional Resources The following pages provide complete documentation for geospatial indexes and queries:

[2dsphere Indexes \(page 324\)](#) A `2dsphere` index supports queries that calculate geometries on an earth-like sphere. The index supports data stored as both GeoJSON objects and as legacy coordinate pairs.

[2d Indexes \(page 325\)](#) The `2d` index supports data stored as legacy coordinate pairs and is intended for use in MongoDB 2.2 and earlier.

[Haystack Indexes \(page 326\)](#) A haystack index is a special index optimized to return results over small areas. For queries that use spherical geometry, a `2dsphere` index is a better option than a haystack index.

[2d Index Internals \(page 327\)](#) Provides a more in-depth explanation of the internals of geospatial indexes. This material is not necessary for normal operations but may be useful for troubleshooting and for further understanding.

2dsphere Indexes New in version 2.4.

A `2dsphere` index supports queries that calculate geometries on an earth-like sphere. The index supports data stored as both *GeoJSON* objects and as legacy coordinate pairs. The index supports legacy coordinate pairs by converting the data to the *GeoJSON Point* type.

The `2dsphere` index supports all MongoDB geospatial queries: queries for inclusion, intersection and proximity.

A [compound](#) (page 318) `2dsphere` index can reference multiple location and non-location fields within a collection's documents. You can arrange the fields in any order.

The default datum for an earth-like sphere in MongoDB 2.4 is *WGS84*. Coordinate-axis order is **longitude, latitude**.

Important: MongoDB allows *only one* geospatial index per collection. You can create either a `2dsphere` or a `2d` (page 325) per collection.

Important: You cannot use a `2dsphere` index as a shard key when sharding a collection. However, you can create and maintain a geospatial index on a sharded collection by using a different field as the shard key.

Store GeoJSON Objects New in version 2.4.

MongoDB supports the following GeoJSON objects:

- *Point*
- *LineString*
- *Polygon*

In order to index GeoJSON data, you must store the data in a location field that you name. The location field contains a subdocument with a `type` field specifying the GeoJSON object type and a `coordinates` field specifying the object's coordinates. Always store coordinates `longitude, latitude` order.

Use the following syntax:

```
{ <location field> : { type : "<GeoJSON type>" ,  
                      coordinates : <coordinates>  
} }
```

The following example stores a GeoJSON Point:

```
{ loc : { type : "Point" ,
          coordinates : [ 40, 5 ]
        } }
```

The following example stores a GeoJSON LineString:

```
{ loc : { type : "LineString" ,
          coordinates : [ [ 40 , 5 ] , [ 41 , 6 ] ]
        } }
```

Polygons consist of an array of GeoJSON LinearRing coordinate arrays. These LinearRings are closed LineStrings. Closed LineStrings have at least four coordinate pairs and specify the same position as the first and last coordinates.

The following example stores a GeoJSON Polygon with an exterior ring and no interior rings (or holes). Note the first and last coordinate pair with the [0 , 0] coordinate:

```
{ loc :
  { type : "Polygon" ,
    coordinates : [ [ [ 0 , 0 ] , [ 3 , 6 ] , [ 6 , 1 ] , [ 0 , 0 ] ] ]
  } }
```

For Polygons with multiple rings:

- The first described ring must be the exterior ring.
- The exterior ring cannot self-intersect.
- Any interior ring must be entirely contained by the outer ring.
- Interior rings cannot intersect or overlap each other. Interior rings can share an edge.

The following document represents a polygon with an interior ring as GeoJSON:

```
{ loc :
  { type : "Polygon" ,
    coordinates : [ [ [ 0 , 0 ] , [ 3 , 6 ] , [ 6 , 1 ] , [ 0 , 0 ] ],
                    [ [ 2 , 2 ] , [ 3 , 3 ] , [ 4 , 2 ] , [ 2 , 2 ] ] ]
  } }
```

2d Indexes

Important: MongoDB allows *only one* geospatial index per collection. You can create either a `2d` or a `2dsphere` (page 324) per collection.

Use a `2d` index for data stored as points on a two-dimensional plane. The `2d` index is intended for legacy coordinate pairs used in MongoDB 2.2 and earlier.

Use a `2d` index if:

- your database has legacy location data from MongoDB 2.2 or earlier, *and*
- you do not intend to store any location data as *GeoJSON* objects.

Do not use a `2d` index if your location data includes GeoJSON objects. To index on both legacy coordinate pairs *and* GeoJSON objects, use a `2dsphere` (page 324) index.

The `2d` index supports calculations on a flat, Euclidean plane. The `2d` index also supports *distance-only* calculations on a sphere, but for *geometric* calculations (e.g. `$geoWithin`) on a sphere, store data as GeoJSON objects and use the `2dsphere` index type.

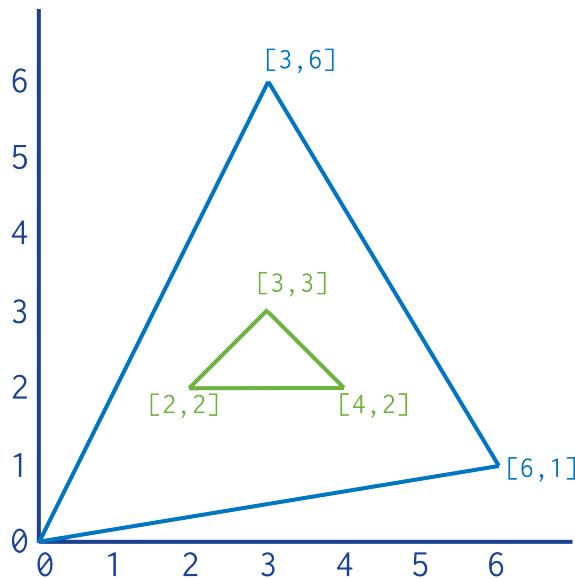


Figure 7.10: Diagram of a Polygon with internal ring.

A 2d index can reference two fields. The first must be the location field. A 2d compound index constructs queries that select first on the location field, and then filters those results by the additional criteria. A compound 2d index can cover queries.

Note: You cannot use a 2d index as a shard key when sharding a collection. However, you can create and maintain a geospatial index on a sharded collection by using a different field as the shard key.

Store Points on a 2D Plane To store location data as legacy coordinate pairs, use either an array (preferred):

```
loc : [ <longitude> , <latitude> ]
```

Or an embedded document:

```
loc : { lng : <longitude> , lat : <latitude> }
```

Arrays are preferred as certain languages do not guarantee associative map ordering.

Whether as an array or document, if you use longitude and latitude, store coordinates in this order: **longitude, latitude**.

Haystack Indexes A haystack index is a special index that is optimized to return results over small areas. Haystack indexes improve performance on queries that use flat geometry.

For queries that use spherical geometry, a **2dsphere index is a better option** than a haystack index. *2dsphere indexes* (page 324) allow field reordering; haystack indexes require the first field to be the location field. Also, haystack indexes are only usable via commands and so always return all results at once.

Haystack indexes create “buckets” of documents from the same geographic area in order to improve performance for queries limited to that area. Each bucket in a haystack index contains all the documents within a specified proximity to a given longitude and latitude.

To create a geohaystacks index, see [Create a Haystack Index](#) (page 351). For information and example on querying a haystack index, see [Query a Haystack Index](#) (page 351).

2d Index Internals This document provides a more in-depth explanation of the internals of MongoDB's 2d geospatial indexes. This material is not necessary for normal operations or application development but may be useful for troubleshooting and for further understanding.

Calculation of Geohash Values for 2d Indexes When you create a geospatial index on *legacy coordinate pairs*, MongoDB computes *geohash* values for the coordinate pairs within the specified [location range](#) (page 348) and then indexes the geohash values.

To calculate a geohash value, recursively divide a two-dimensional map into quadrants. Then assign each quadrant a two-bit value. For example, a two-bit representation of four quadrants would be:

```
01 11
```

```
00 10
```

These two-bit values (00, 01, 10, and 11) represent each of the quadrants and all points within each quadrant. For a geohash with two bits of resolution, all points in the bottom left quadrant would have a geohash of 00. The top left quadrant would have the geohash of 01. The bottom right and top right would have a geohash of 10 and 11, respectively.

To provide additional precision, continue dividing each quadrant into sub-quadrants. Each sub-quadrant would have the geohash value of the containing quadrant concatenated with the value of the sub-quadrant. The geohash for the upper-right quadrant is 11, and the geohash for the sub-quadrants would be (clockwise from the top left): 1101, 1111, 1110, and 1100, respectively.

Multi-location Documents for 2d Indexes

New in version 2.0: Support for multiple locations in a document.

While 2d geospatial indexes do not support more than one set of coordinates in a document, you can use a [multi-key index](#) (page 320) to index multiple coordinate pairs in a single document. In the simplest example you may have a field (e.g. `locs`) that holds an array of coordinates, as in the following example:

```
{ _id : ObjectId(...),
  locs : [ [ 55.5 , 42.3 ] ,
            [ -74 , 44.74 ] ,
            { lng : 55.5 , lat : 42.3 } ]
}
```

The values of the array may be either arrays, as in `[55.5, 42.3]`, or embedded documents, as in `{ lng : 55.5 , lat : 42.3 }`.

You could then create a geospatial index on the `locs` field, as in the following:

```
db.places.ensureIndex( { "locs": "2d" } )
```

You may also model the location data as a field inside of a sub-document. In this case, the document would contain a field (e.g. `addresses`) that holds an array of documents where each document has a field (e.g. `loc`) that holds location coordinates. For example:

```
{ _id : ObjectId(...),
  name : "...",
  addresses : [ {
      context : "home" ,
      loc : [ 55.5 , 42.3 ]
    } ,
    ...
  ]
}
```

```
        {
          context : "home",
          loc : [ -74 , 44.74 ]
        }
      ]
}
```

You could then create the geospatial index on the `addresses.loc` field as in the following example:

```
db.records.ensureIndex( { "addresses.loc": "2d" } )
```

For documents with multiple coordinate values, queries may return the same document multiple times if more than one indexed coordinate pair satisfies the query constraints. Use the `uniqueDocs` parameter to `geoNear` or the `$uniqueDocs` operator with `$geoWithin`.

To include the location field with the distance field in multi-location document queries, specify `includeLocs: true` in the `geoNear` command.

See also:

geospatial-query-compatibility-chart

Text Indexes

New in version 2.4.

MongoDB provides `text` indexes to support text search of string content in documents of a collection. `text` indexes are case-insensitive and can include any field whose value is a string or an array of string elements. You can only access the `text` index with the `text` command.

Important:

- Before you can create a text index or [run the text command](#) (page 329), you need to manually enable the text search. See [Enable Text Search](#) (page 354) for information on how to enable the text search feature.
 - A collection can have at most **one** `text` index.
-

Create Text Index To create a `text` index, use the `db.collection.ensureIndex()` method. To index a field that contains a string or an array of string elements, include the field and specify the string literal `"text"` in the index document, as in the following example:

```
db.reviews.ensureIndex( { comments: "text" } )
```

For examples of creating `text` indexes on multiple fields, see [Create a text Index](#) (page 355).

`text` indexes drop language-specific stop words (e.g. in English, “the,” “an,” “a,” “and,” etc.) and uses simple language-specific suffix stemming. See [text-search-languages](#) for the supported languages and [Specify a Language for Text Index](#) (page 358) for details on specifying languages with `text` indexes.

`text` indexes can cover a text search. If an index covers a text search, MongoDB does not need to inspect data outside of the index to fulfill the search. For details, see [Create text Index to Cover Queries](#) (page 363).

Storage Requirements and Performance Costs `text` indexes have the following storage requirements and performance costs:

- `text` indexes change the space allocation method for all future record allocations in a collection to `usePowerOf2Sizes`.

- text indexes can be large. They contain one index entry for each unique post-stemmed word in each indexed field for each document inserted.
- Building a text index is very similar to building a large multi-key index and will take longer than building a simple ordered (scalar) index on the same data.
- When building a large text index on an existing collection, ensure that you have a sufficiently high limit on open file descriptors. See the [recommended settings](#) (page 219).
- text indexes will impact insertion throughput because MongoDB must add an index entry for each unique post-stemmed word in each indexed field of each new source document.
- Additionally, text indexes do not store phrases or information about the proximity of words in the documents. As a result, phrase queries will run much more effectively when the entire collection fits in RAM.

Text Search Text search supports the search of string content in documents of a collection. MongoDB provides the `text` command to perform the text search. The `text` command accesses the `text` index.

The text search process:

- tokenizes and stems the search term(s) during both the index creation and the `text` command execution.
- assigns a score to each document that contains the search term in the indexed fields. The score determines the relevance of a document to a given search query.

By default, the `text` command returns at most the top 100 matching documents as determined by the scores. The command can search for words and phrases. The command matches on the complete stemmed words. For example, if a document field contains the word `blueberry`, a search on the term `blue` will not match the document. However, a search on either `blueberry` or `blueberries` will match.

For information and examples on various text search patterns, see [Search String Content for Text](#) (page 355).

Hashed Index

New in version 2.4.

Hashed indexes maintain entries with hashes of the values of the indexed field. The hashing function collapses sub-documents and computes the hash for the entire value but does not support multi-key (i.e. arrays) indexes.

Hashed indexes support [sharding](#) (page 487) a collection using a [hashed shard key](#) (page 500). Using a hashed shard key to shard a collection ensures a more even distribution of data. See [Shard a Collection Using a Hashed Shard Key](#) (page 520) for more details.

MongoDB can use the hashed index to support equality queries, but hashed indexes do not support range queries.

You may not create compound indexes that have hashed index fields or specify a unique constraint on a hashed index; however, you can create both a hashed index and an ascending/descending (i.e. non-hashed) index on the same field: MongoDB will use the scalar index for range queries.

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 consistently converted to 64-bit integers (and then back to floating point). MongoDB hashed indexes do not support floating point values larger than 2^{53} .

Create a hashed index using an operation that resembles the following:

```
db.active.ensureIndex( { a: "hashed" } )
```

This operation creates a hashed index for the `active` collection on the `a` field.

7.2.2 Index Properties

In addition to the numerous [index types](#) (page 315) MongoDB supports, indexes can also have various properties. The following documents detail the index properties that can you can select when building an index.

TTL Indexes (page 330) The TTL index is used for TTL collections, which expire data after a period of time.

Unique Indexes (page 330) A unique index causes MongoDB to reject all documents that contain a duplicate value for the indexed field.

Sparse Indexes (page 331) A sparse index does not index documents that do not have the indexed field.

TTL Indexes

TTL indexes are special indexes that MongoDB can use to automatically remove documents from a collection after a certain amount of time. This is ideal for some types of information like machine generated event data, logs, and session information that only need to persist in a database for a limited amount of time.

These indexes have the following limitations:

- [Compound indexes](#) (page 318) are *not* supported.
- The indexed field **must** be a date *type*.
- If the field holds an array, and there are multiple date-typed data in the index, the document will expire when the *lowest* (i.e. earliest) matches the expiration threshold.

Warning: The TTL index does not guarantee that expired data will be deleted immediately. There may be a delay between the time a document expires and the time that MongoDB removes the document from the database.

The background task that removes expired documents runs *every 60 seconds*. As a result, documents may remain in a collection *after* they expire but *before* the background task runs or completes.

The duration of the removal operation depends on the workload of your `mongod` instance. Therefore, expired data may exist for some time *beyond* the 60 second period between runs of the background task.

In all other respects, TTL indexes are normal indexes, and if appropriate, MongoDB can use these indexes to fulfill arbitrary queries.

See

[Expire Data from Collections by Setting TTL](#) (page 160)

Unique Indexes

A unique index causes MongoDB to reject all documents that contain a duplicate value for the indexed field. To create a unique index on the `user_id` field of the `members` collection, use the following operation in the `mongo` shell:

```
db.addresses.ensureIndex( { "user_id": 1 }, { unique: true } )
```

By default, `unique` is `false` on MongoDB indexes.

If you use the `unique` constraint on a [compound index](#) (page 318), then MongoDB will enforce uniqueness on the *combination* of values rather than the individual value for any or all values of the key.

If a document does not have a value for the indexed field in a unique index, the index will store a null value for this document. Because of the unique constraint, MongoDB will only permit one document that lacks the indexed field. If there is more than one document without a value for the indexed field or is missing the indexed field, the index build will fail with a duplicate key error.

You can combine the unique constraint with the [sparse index](#) (page 331) to filter these null values from the unique index and avoid the error.

You may not specify a unique constraint on a [hashed index](#) (page 329).

Sparse Indexes

Sparse indexes only contain entries for documents that have the indexed field, even if the index field contains a null value. The index skips over any document that is missing the indexed field. The index is “sparse” because it does not include all documents of a collection. By contrast, non-sparse indexes contain all documents in a collection, storing null values for those documents that do not contain the indexed field.

The following example in the mongo shell creates a sparse index on the `xmpp_id` field of the `members` collection:

```
db.addresses.ensureIndex( { "xmpp_id": 1 }, { sparse: true } )
```

By default, `sparse` is `false` on MongoDB indexes.

Warning: Using these indexes will sometimes result in incomplete results when filtering or sorting results, because sparse indexes are not complete for all documents in a collection.

Note: Do not confuse sparse indexes in MongoDB with [block-level](#)³ indexes in other databases. Think of them as dense indexes with a specific filter.

Tip

You can specify a `sparse` and [unique indexes](#) (page 330), that rejects documents that have duplicate values for a field, but allows multiple documents that omit that key.

Examples

Sparse Index On A Collection Can Results In Incomplete Results Consider a collection `scores` that contains the following documents:

```
{ "_id" : ObjectId("523b6e32fb408eea0eec2647"), "userid" : "newbie" }
{ "_id" : ObjectId("523b6e61fb408eea0eec2648"), "userid" : "abby", "score" : 82 }
{ "_id" : ObjectId("523b6e6ffb408eea0eec2649"), "userid" : "nina", "score" : 90 }
```

The collection has a sparse index on the field `score`:

```
db.scores.ensureIndex( { score: 1 }, { sparse: true } )
```

Then, the following query to return all documents in the `scores` collection sorted by the `score` field results in incomplete results:

```
db.scores.find().sort( { score: -1 } )
```

Because the document for the `userid` “`newbie`” does not contain the `score` field, the query, which uses the sparse index, will return incomplete results that omit that document:

```
{ "_id" : ObjectId("523b6e6ffb408eea0eec2649"), "userid" : "nina", "score" : 90 }
{ "_id" : ObjectId("523b6e61fb408eea0eec2648"), "userid" : "abby", "score" : 82 }
```

³http://en.wikipedia.org/wiki/Database_index#Sparse_index

Sparse Index with Unique Constraint Consider a collection `scores` that contains the following documents:

```
{ "_id" : ObjectId("523b6e32fb408eea0eec2647"), "userid" : "newbie" }
{ "_id" : ObjectId("523b6e61fb408eea0eec2648"), "userid" : "abby", "score" : 82 }
{ "_id" : ObjectId("523b6e6ffb408eea0eec2649"), "userid" : "nina", "score" : 90 }
```

You could create an index with a [unique constraint](#) (page 330) and sparse filter on the `score` field using the following operation:

```
db.scores.ensureIndex( { score: 1 } , { sparse: true, unique: true } )
```

This index *would permit* inserting documents that had unique values for the `score` field *or* did not include a `score` field. Consider the following [insert operation](#) (page 56):

```
db.scores.insert( { "userid": "PWWfO8lFs1", "score": 43 } )
db.scores.insert( { "userid": "XlSOX66gEy", "score": 34 } )
db.scores.insert( { "userid": "nuZHu2tcRm" } )
db.scores.insert( { "userid": "HIGvEZfdc5" } )
```

However, this index *would not permit* adding the following documents:

```
db.scores.insert( { "userid": "PWWfO8lFs1", "score": 82 } )
db.scores.insert( { "userid": "XlSOX66gEy", "score": 90 } )
```

7.2.3 Index Creation

MongoDB provides several options that *only* affect the creation of the index. Specify these options in a document as the second argument to the `db.collection.ensureIndex()` method. This section describes the uses of these creation options and their behavior.

Related

Some options that you can specify to `ensureIndex()` options control the [properties of the index](#) (page 330), which are *not* index creation options. For example, the [unique](#) (page 330) option affects the behavior of the index after creation.

For a detailed description of MongoDB's index types, see [Index Types](#) (page 315) and [Index Properties](#) (page 330) for related documentation.

Background Construction

By default, creating an index blocks all other operations on a database. When building an index on a collection, the database that holds the collection is unavailable for read or write operations until the index build completes. Any operation that requires a read or write lock on all databases (e.g. `listDatabases`) will wait for the foreground index build to complete.

For potentially long running index building operations, consider the `background` operation so that the MongoDB database remains available during the index building operation. For example, to create an index in the background of the `zipcode` field of the `people` collection, issue the following:

```
db.people.ensureIndex( { zipcode: 1 }, {background: true} )
```

By default, `background` is `false` for building MongoDB indexes.

You can combine the `background` option with other options, as in the following:

```
db.people.ensureIndex( { zipcode: 1 }, {background: true, sparse: true} )
```

Behavior

As of MongoDB version 2.4, a mongod instance can build more than one index in the background concurrently.

Changed in version 2.4: Before 2.4, a mongod instance could only build one background index per database at a time.

Changed in version 2.2: Before 2.2, a single mongod instance could only build one index at a time.

Background indexing operations run in the background so that other database operations can run while creating the index. However, the mongo shell session or connection where you are creating the index *will* block until the index build is complete. To continue issuing commands to the database, open another connection or mongo instance.

Queries will not use partially-built indexes: the index will only be usable once the index build is complete.

Note: If MongoDB is building an index in the background, you cannot perform other administrative operations involving that collection, including running repairDatabase, dropping the collection (i.e. db.collection.drop()), and running compact. These operations will return an error during background index builds.

Performance

The background index operation uses an incremental approach that is slower than the normal “foreground” index builds. If the index is larger than the available RAM, then the incremental process can take *much* longer than the foreground build.

If your application includes ensureIndex() operations, and an index *doesn't* exist for other operational concerns, building the index can have a severe impact on the performance of the database.

To avoid performance issues, make sure that your application checks for the indexes at start up using the getIndexes() method or the equivalent method for your driver⁴ and terminates if the proper indexes do not exist. Always build indexes in production instances using separate application code, during designated maintenance windows.

Building Indexes on Secondaries

Background index operations on a *replica set primary* become foreground indexing operations on *secondary members* of the set. All indexing operations on secondaries block replication.

To build large indexes on secondaries the best approach is to restart one secondary at a time in *standalone* mode and build the index. After building the index, restart as a member of the replica set, allow it to catch up with the other members of the set, and then build the index on the next secondary. When all the secondaries have the new index, step down the primary, restart it as a standalone, and build the index on the former primary.

Remember, the amount of time required to build the index on a secondary must be within the window of the *oplog*, so that the secondary can catch up with the primary.

Indexes on secondary members in “recovering” mode are always built in the foreground to allow them to catch up as soon as possible.

See [Build Indexes on Replica Sets](#) (page 339) for a complete procedure for building indexes on secondaries.

⁴<http://api.mongodb.org/>

Drop Duplicates

MongoDB cannot create a [unique index](#) (page 330) on a field that has duplicate values. To force the creation of a unique index, you can specify the `dropDups` option, which will only index the first occurrence of a value for the key, and delete all subsequent values.

Important: As in all unique indexes, if a document does not have the indexed field, MongoDB will include it in the index with a “null” value.

If subsequent fields *do not* have the indexed field, and you have set `{dropDups: true}`, MongoDB will remove these documents from the collection when creating the index. If you combine `dropDups` with the [sparse](#) (page 331) option, this index will only include documents in the index that have the value, and the documents without the field will remain in the database.

To create a unique index that drops duplicates on the `username` field of the `accounts` collection, use a command in the following form:

```
db.accounts.ensureIndex( { username: 1 }, { unique: true, dropDups: true } )
```

Warning: Specifying `{ dropDups: true }` will delete data from your database. Use with extreme caution.

By default, `dropDups` is `false`.

Index Names

The default name for an index is the concatenation of the indexed keys and each key’s direction in the index, `1` or `-1`.

Example

Issue the following command to create an index on `item` and `quantity`:

```
db.products.ensureIndex( { item: 1, quantity: -1 } )
```

The resulting index is named: `item_1_quantity_-1`.

Optionally, you can specify a name for an index instead of using the default name.

Example

Issue the following command to create an index on `item` and `quantity` and specify `inventory` as the index name:

```
db.products.ensureIndex( { item: 1, quantity: -1 } , { name: "inventory" } )
```

The resulting index has the name `inventory`.

To view the name of an index, use the `getIndexes()` method.

7.3 Indexing Tutorials

Indexes allow MongoDB to process and fulfill queries quickly by creating small and efficient representations of the documents in a collection.

The documents in this section outline specific tasks related to building and maintaining indexes for data in MongoDB collections and discusses strategies and practical approaches. For a conceptual overview of MongoDB indexing, see the [Index Concepts](#) (page 314) document.

[Index Creation Tutorials](#) (page 335) Create and configure different types of indexes for different purposes.

[Index Management Tutorials](#) (page 342) Monitor and assess index performance and rebuild indexes as needed.

[Geospatial Index Tutorials](#) (page 345) Create indexes that support data stored as *GeoJSON* objects and legacy coordinate pairs.

[Text Search Tutorials](#) (page 354) Build and configure indexes that support full-text searches.

[Indexing Strategies](#) (page 363) The factors that affect index performance and practical approaches to indexing in MongoDB

7.3.1 Index Creation Tutorials

Instructions for creating and configuring indexes in MongoDB and building indexes on replica sets and sharded clusters.

[Create an Index](#) (page 335) Build an index for any field on a collection.

[Create a Compound Index](#) (page 336) Build an index of multiple fields on a collection.

[Create a Unique Index](#) (page 337) Build an index that enforces unique values for the indexed field or fields.

[Create a Sparse Index](#) (page 338) Build an index that omits references to documents that do not include the indexed field. This saves space when indexing fields that are present in only some documents.

[Create a Hashed Index](#) (page 338) Compute a hash of the value of a field in a collection and index the hashed value. These indexes permit equality queries and may be suitable shard keys for some collections.

[Build Indexes on Replica Sets](#) (page 339) To build indexes on a replica set, you build the indexes separately on the primary and the secondaries, as described here.

[Build Indexes in the Background](#) (page 341) Background index construction allows read and write operations to continue while building the index, but take longer to complete and result in a larger index.

[Build Old Style Indexes](#) (page 341) A `{ v : 0 }` index is necessary if you need to roll back from MongoDB version 2.0 (or later) to MongoDB version 1.8.

Create an Index

Indexes allow MongoDB to process and fulfill queries quickly by creating small and efficient representations of the documents in a *collection*. MongoDB creates an index on the `_id` field of every collection by default, but allows users to create indexes for any collection using on any field in a *document*.

This tutorial describes how to create an index on a single field. MongoDB also supports [compound indexes](#) (page 318), which are indexes on multiple fields. See [Create a Compound Index](#) (page 336) for instructions on building compound indexes.

Create an Index on a Single Field

To create an index, use `ensureIndex()` or a similar method from your driver⁵. For example the following creates an index on the `phone-number` field of the `people` collection:

⁵<http://api.mongodb.org/>

```
db.people.ensureIndex( { "phone-number": 1 } )
```

`ensureIndex()` only creates an index if an index of the same specification does not already exist.

All indexes support and optimize the performance for queries that select on this field. For queries that cannot use an index, MongoDB must scan all documents in a collection for documents that match the query.

Examples

If you create an index on the `user_id` field in the `records`, this index is, the index will support the following query:

```
db.records.find( { user_id: 2 } )
```

However, the following query, on the `profile_url` field is not supported by this index:

```
db.records.find( { profile_url: 2 } )
```

Additional Considerations

If your collection holds a large amount of data, and your application needs to be able to access the data while building the index, consider building the index in the background, as described in [Background Construction](#) (page 332). To build indexes on replica sets, see the [Build Indexes on Replica Sets](#) (page 339) section for more information.

Note: To build or rebuild indexes for a *replica set* see [Build Indexes on Replica Sets](#) (page 339).

Some drivers may specify indexes, using `NumberLong(1)` rather than `1` as the specification. This does not have any affect on the resulting index.

See also:

[Create a Compound Index](#) (page 336), [Indexing Tutorials](#) (page 334) and [Index Concepts](#) (page 314) for more information.

Create a Compound Index

Indexes allow MongoDB to process and fulfill queries quickly by creating small and efficient representations of the documents in a *collection*. MongoDB supports indexes that include content on a single field, as well as [compound indexes](#) (page 318) that include content from multiple fields. Continue reading for instructions and examples of building a compound index.

Build a Compound Index

To create a [compound index](#) (page 318) use an operation that resembles the following prototype:

```
db.collection.ensureIndex( { a: 1, b: 1, c: 1 } )
```

Example

The following operation will create an index on the `item`, `category`, and `price` fields of the `products` collection:

```
db.products.ensureIndex( { item: 1, category: 1, price: 1 } )
```

Additional Considerations

If your collection holds a large amount of data, and your application needs to be able to access the data while building the index, consider building the index in the background, as described in [Background Construction](#) (page 332). To build indexes on replica sets, see the [Build Indexes on Replica Sets](#) (page 339) section for more information.

Note: To build or rebuild indexes for a *replica set* see [Build Indexes on Replica Sets](#) (page 339).

Some drivers may specify indexes, using `NumberLong(1)` rather than `1` as the specification. This does not have any affect on the resulting index.

See also:

[Create an Index](#) (page 335), [Indexing Tutorials](#) (page 334) and [Index Concepts](#) (page 314) for more information.

Create a Unique Index

MongoDB allows you to specify a [*unique constraint*](#) (page 330) on an index. These constraints prevent applications from inserting *documents* that have duplicate values for the inserted fields. Additionally, if you want to create an index on a collection that has existing data that might have duplicate values for the indexed field, you may chose combine unique enforcement with [*duplicate dropping*](#) (page 334).

Unique Indexes

To create a [*unique indexes*](#) (page 330), consider the following prototype:

```
db.collection.ensureIndex( { a: 1 }, { unique: true } )
```

For example, you may want to create a unique index on the `"tax-id":` of the `accounts` collection to prevent storing multiple account records for the same legal entity:

```
db.accounts.ensureIndex( { "tax-id": 1 }, { unique: true } )
```

The [*_id index*](#) (page 317) is a unique index. In some situations you may consider using `_id` field itself for this kind of data rather than using a unique index on another field.

In many situations you will want to combine the `unique` constraint with the `sparse` option. When MongoDB indexes a field, if a document does not have a value for a field, the index entry for that item will be `null`. Since unique indexes cannot have duplicate values for a field, without the `sparse` option, MongoDB will reject the second document and all subsequent documents without the indexed field. Consider the following prototype.

```
db.collection.ensureIndex( { a: 1 }, { unique: true, sparse: true } )
```

You can also enforce a unique constraint on [*compound indexes*](#) (page 318), as in the following prototype:

```
db.collection.ensureIndex( { a: 1, b: 1 }, { unique: true } )
```

These indexes enforce uniqueness for the *combination* of index keys and *not* for either key individually.

Drop Duplicates

To force the creation of a [unique index](#) (page 330) index on a collection with duplicate values in the field you are indexing you can use the `dropDups` option. This will force MongoDB to create a [unique](#) index by deleting documents with duplicate values when building the index. Consider the following prototype invocation of `ensureIndex()`:

```
db.collection.ensureIndex( { a: 1 }, { unique: true, dropDups: true } )
```

See the full documentation of [duplicate dropping](#) (page 334) for more information.

Warning: Specifying `{ dropDups: true }` may delete data from your database. Use with extreme caution.

Refer to the `ensureIndex()` documentation for additional index creation options.

Create a Sparse Index

Sparse indexes are like non-sparse indexes, except that they omit references to documents that do not include the indexed field. For fields that are only present in some documents sparse indexes may provide a significant space savings. See [Sparse Indexes](#) (page 331) for more information about sparse indexes and their use.

See also:

[Index Concepts](#) (page 314) and [Indexing Tutorials](#) (page 334) for more information.

Prototype

To create a [sparse index](#) (page 331) on a field, use an operation that resembles the following prototype:

```
db.collection.ensureIndex( { a: 1 }, { sparse: true } )
```

Example

The following operation, creates a sparse index on the `users` collection that *only* includes a document in the index if the `twitter_name` field exists in a document.

```
db.users.ensureIndex( { twitter_name: 1 }, { sparse: true } )
```

The index excludes all documents that do not include the `twitter_name` field.

Considerations

Note: Sparse indexes can affect the results returned by the query, particularly with respect to sorts on fields *not* included in the index. See the [sparse index](#) (page 331) section for more information.

Create a Hashed Index

New in version 2.4.

[Hashed indexes](#) (page 329) compute a hash of the value of a field in a collection and index the hashed value. These indexes permit equality queries and may be suitable shard keys for some collections.

Tip

MongoDB automatically computes the hashes when resolving queries using hashed indexes. Applications do **not** need to compute hashes.

See

[Hashed Shard Keys](#) (page 500) for more information about hashed indexes in sharded clusters, as well as [Index Concepts](#) (page 314) and [Indexing Tutorials](#) (page 334) for more information about indexes.

Procedure

To create a [hashed index](#) (page 329), specify `hashed` as the value of the index key, as in the following example:

Example

Specify a hashed index on `_id`

```
db.collection.ensureIndex( { _id: "hashed" } )
```

Considerations

MongoDB supports `hashed` indexes of any single field. The hashing function collapses sub-documents and computes the hash for the entire value, but does not support multi-key (i.e. arrays) indexes.

You may not create compound indexes that have `hashed` index fields.

Build Indexes on Replica Sets

[Background index creation operations](#) (page 332) become *foreground* indexing operations on *secondary* members of replica sets. The foreground index building process blocks all replication and read operations on the secondaries while they build the index.

Secondaries will begin building indexes *after* the *primary* finishes building the index. In *sharded clusters*, the `mongos` will send `ensureIndex()` to the primary members of the replica set for each shard, which then replicate to the secondaries after the primary finishes building the index.

To minimize the impact of building an index on your replica set, use the following procedure to build indexes on secondaries:

See

[Indexing Tutorials](#) (page 334) and [Index Concepts](#) (page 314) for more information.

Considerations

Warning: Ensure that your `oplog` is large enough to permit the indexing or re-indexing operation to complete without falling too far behind to catch up. See the [oplog sizing](#) (page 406) documentation for additional information.

Note: This procedure *does* take one member out of the replica set at a time. However, this procedure will only affect one member of the set at a time rather than *all* secondaries at the same time.

Procedure

Note: If you need to build an index in a *sharded cluster*, repeat the following procedure for each replica set that provides each *shard*.

Stop One Secondary Stop the mongod process on one secondary. Restart the mongod process *without* the `--rep1Set` option and running on a different port.⁶ This instance is now in “standalone” mode.

For example, if your mongod *normally* runs with on the default port of 27017 with the `--rep1Set` option you would use the following invocation:

```
mongod --port 47017
```

Build the Index Create the new index using the `ensureIndex()` in the mongo shell, or comparable method in your driver. This operation will create or rebuild the index on this mongod instance

For example, to create an ascending index on the `username` field of the `records` collection, use the following mongo shell operation:

```
db.records.ensureIndex( { username: 1 } )
```

See also:

[Create an Index](#) (page 335) and [Create a Compound Index](#) (page 336) for more information.

Restart the Program mongod When the index build completes, start the mongod instance with the `--rep1Set` option on its usual port:

```
mongod --port 27017 --rep1Set rs0
```

Modify the port number (e.g. 27017) or the replica set name (e.g. `rs0`) as needed.

Allow replication to catch up on this member.

Build Indexes on all Secondaries For each secondary in the set, build an index according to the following steps:

1. [Stop One Secondary](#) (page 340)
2. [Build the Index](#) (page 340)
3. [Restart the Program mongod](#) (page 340)

⁶ By running the mongod on a different port, you ensure that the other members of the replica set and all clients will not contact the member while you are building the index.

Build the Index on the Primary To build an index on the primary you can either:

1. [Build the index in the background](#) (page 341) on the primary.
2. Step down the primary using the method:`rs.stepDown()` method in the `mongo` shell to cause the current primary to become a secondary graceful and allow the set to elect another member as primary.

Then repeat the index building procedure, listed below, to build the index on the primary:

- (a) [Stop One Secondary](#) (page 340)
- (b) [Build the Index](#) (page 340)
- (c) [Restart the Program `mongod`](#) (page 340)

Building the index on the background, takes longer than the foreground index build and results in a less compact index structure. Additionally, the background index build may impact write performance on the primary. However, building the index in the background allows the set to be continuously up for write operations during while MongoDB builds the index.

Build Indexes in the Background

By default, MongoDB builds indexes in the foreground and prevent all read and write operations to the database while the index builds. Also, no operation that requires a read or write lock on all databases (e.g. `listDatabases`) can occur during a foreground index build.

[Background index construction](#) (page 332) allows read and write operations to continue while building the index.

Note: Background index builds take longer to complete and result in a larger index.

After the index finishes building, MongoDB treats indexes built in the background the same as any other index.

See also:

[Index Concepts](#) (page 314) and [Indexing Tutorials](#) (page 334) for more information.

Procedure

To create an index in the background, add the `background` argument to the `ensureIndex()` operation, as in the following index:

```
db.collection.ensureIndex( { a: 1 }, { background: true } )
```

Consider the section on [background index construction](#) (page 332) for more information about these indexes and their implications.

Build Old Style Indexes

Important: Use this procedure *only* if you **must** have indexes that are compatible with a version of MongoDB earlier than 2.0.

MongoDB version 2.0 introduced the `{v:1}` index format. MongoDB versions 2.0 and later support both the `{v:1}` format and the earlier `{v:0}` format.

MongoDB versions prior to 2.0, however, support only the `{v:0}` format. If you need to roll back MongoDB to a version prior to 2.0, you must *drop* and *re-create* your indexes.

To build pre-2.0 indexes, use the `dropIndexes()` and `ensureIndex()` methods. You *cannot* simply reindex the collection. When you reindex on versions that only support `{v:0}` indexes, the `v` fields in the index definition still hold values of 1, even though the indexes would now use the `{v:0}` format. If you were to upgrade again to version 2.0 or later, these indexes would not work.

Example

Suppose you rolled back from MongoDB 2.0 to MongoDB 1.8, and suppose you had the following index on the `items` collection:

```
{ "v" : 1, "key" : { "name" : 1 }, "ns" : "mydb.items", "name" : "name_1" }
```

The `v` field tells you the index is a `{v:1}` index, which is incompatible with version 1.8.

To drop the index, issue the following command:

```
db.items.dropIndex( { name : 1 } )
```

To recreate the index as a `{v:0}` index, issue the following command:

```
db.foo.ensureIndex( { name : 1 } , { v : 0 } )
```

See also:

[Index Performance Enhancements](#) (page 640).

7.3.2 Index Management Tutorials

Instructions for managing indexes and assessing index performance and use.

[Remove Indexes](#) (page 342) Drop an index from a collection.

[Rebuild Indexes](#) (page 343) In a single operation, drop all indexes on a collection and then rebuild them.

[Manage In-Progress Index Creation](#) (page 344) Check the status of indexing progress, or terminate an ongoing index build.

[Return a List of All Indexes](#) (page 344) Obtain a list of all indexes on a collection or of all indexes on all collections in a database.

[Measure Index Use](#) (page 344) Study query operations and observe index use for your database.

Remove Indexes

To remove an index from a collection use the `dropIndex()` method and the following procedure. If you simply need to rebuild indexes you can use the process described in the [Rebuild Indexes](#) (page 343) document.

See also:

[Indexing Tutorials](#) (page 334) and [Index Concepts](#) (page 314) for more information about indexes and indexing operations in MongoDB.

Operations

To remove an index, use the `db.collection.dropIndex()` method, as in the following example:

```
db.accounts.dropIndex( { "tax-id": 1 } )
```

This will remove the index on the "tax-id" field in the `accounts` collection. The shell provides the following document after completing the operation:

```
{ "nIndexesWas" : 3, "ok" : 1 }
```

Where the value of `nIndexesWas` reflects the number of indexes *before* removing this index. You can also use the `db.collection.dropIndexes()` to remove *all* indexes, except for the `_id index` (page 317) from a collection.

These shell helpers provide wrappers around the `dropIndexes` database command. Your `client library` (page 92) may have a different or additional interface for these operations.

Rebuild Indexes

If you need to rebuild indexes for a collection you can use the `db.collection.reIndex()` method to rebuild all indexes on a collection in a single operation. This operation drops all indexes, including the `_id index` (page 317), and then rebuilds all indexes.

See also:

[Index Concepts](#) (page 314) and [Indexing Tutorials](#) (page 334).

Process

The operation takes the following form:

```
db.accounts.reIndex()
```

MongoDB will return the following document when the operation completes:

```
{
  "nIndexesWas" : 2,
  "msg" : "indexes dropped for collection",
  "nIndexes" : 2,
  "indexes" : [
    {
      "key" : {
        "_id" : 1,
        "tax-id" : 1
      },
      "ns" : "records.accounts",
      "name" : "_id_"
    }
  ],
  "ok" : 1
}
```

This shell helper provides a wrapper around the `reIndex` database command. Your `client library` (page 92) may have a different or additional interface for this operation.

Additional Considerations

Note: To build or rebuild indexes for a *replica set* see [Build Indexes on Replica Sets](#) (page 339).

Manage In-Progress Index Creation

To see the status of the indexing processes, you can use the `db.currentOp()` method in the mongo shell. The value of the `query` field and the `msg` field will indicate if the operation is an index build. The `msg` field also indicates the percent of the build that is complete.

To terminate an ongoing index build, use the `db.killOp()` method in the mongo shell.

For more information see `db.currentOp()`.

Changed in version 2.4: Before MongoDB 2.4, you could *only* terminate *background* index builds. After 2.4, you can terminate any index build, including foreground index builds.

Return a List of All Indexes

When performing maintenance you may want to check which indexes exist on a collection. Every index on a collection has a corresponding *document* in the `system.indexes` (page 223) collection, and you can use standard queries (i.e. `find()`) to list the indexes, or in the mongo shell, the `getIndexes()` method to return a list of the indexes on a collection, as in the following examples.

See also:

[Index Concepts](#) (page 314) and [Indexing Tutorials](#) (page 334) for more information about indexes in MongoDB and common index management operations.

List all Indexes on a Collection

To return a list of all indexes on a collection, use the `db.collection.getIndexes()` method or a similar method for your driver⁷.

For example, to view all indexes on the `people` collection:

```
db.people.getIndexes()
```

List all Indexes for a Database

To return a list of all indexes on all collections in a database, use the following operation in the mongo shell:

```
db.system.indexes.find()
```

See `system.indexes` (page 223) for more information about these documents.

Measure Index Use

Synopsis

Query performance is a good general indicator of index use; however, for more precise insight into index use, MongoDB provides a number of tools that allow you to study query operations and observe index use for your database.

See also:

[Index Concepts](#) (page 314) and [Indexing Tutorials](#) (page 334) for more information.

⁷<http://api.mongodb.org/>

Operations

Return Query Plan with `explain()` Append the `explain()` method to any cursor (e.g. query) to return a document with statistics about the query process, including the index used, the number of documents scanned, and the time the query takes to process in milliseconds.

Control Index Use with `hint()` Append the `hint()` to any cursor (e.g. query) with the index as the argument to force MongoDB to use a specific index to fulfill the query. Consider the following example:

```
db.people.find( { name: "John Doe", zipcode: { $gt: 63000 } } ).hint( { zipcode: 1 } )
```

You can use `hint()` and `explain()` in conjunction with each other to compare the effectiveness of a specific index. Specify the `$natural` operator to the `hint()` method to prevent MongoDB from using *any* index:

```
db.people.find( { name: "John Doe", zipcode: { $gt: 63000 } } ).hint( { $natural: 1 } )
```

Instance Index Use Reporting MongoDB provides a number of metrics of index use and operation that you may want to consider when analyzing index use for your database:

- In the output of `serverStatus`:
 - `indexCounters`
 - `scanned`
 - `scanAndOrder`
- In the output of `collStats`:
 - `totalIndexSize`
 - `indexSizes`
- In the output of `dbStats`:
 - `dbStats.indexes`
 - `dbStats.indexSize`

7.3.3 Geospatial Index Tutorials

Instructions for creating and querying 2d, 2dsphere, and haystack indexes.

[Create a 2dsphere Index \(page 346\)](#) A 2dsphere index supports data stored as both GeoJSON objects and as legacy coordinate pairs.

[Query a 2dsphere Index \(page 346\)](#) Search for locations within, near, or intersected by a GeoJSON shape, or within a circle as defined by coordinate points on a sphere.

[Create a 2d Index \(page 348\)](#) Create a 2d index to support queries on data stored as legacy coordinate pairs.

[Query a 2d Index \(page 349\)](#) Search for locations using legacy coordinate pairs.

[Create a Haystack Index \(page 351\)](#) A haystack index is optimized to return results over small areas. For queries that use spherical geometry, a 2dsphere index is a better option.

[Query a Haystack Index \(page 351\)](#) Search based on location and non-location data within a small area.

[Calculate Distance Using Spherical Geometry \(page 352\)](#) Convert distances to radians and back again.

Create a 2dsphere Index

To create a geospatial index for GeoJSON-formatted data, use the `ensureIndex()` method and set the value of the location field for your collection to `2dsphere`. A `2dsphere` index can be a [compound index](#) (page 318) and does not require the location field to be the first field indexed.

To create the index use the following syntax:

```
db.points.ensureIndex( { <location field> : "2dsphere" } )
```

The following are four example commands for creating a `2dsphere` index:

```
db.points.ensureIndex( { loc : "2dsphere" } )
db.points.ensureIndex( { loc : "2dsphere" , type : 1 } )
db.points.ensureIndex( { rating : 1 , loc : "2dsphere" } )
db.points.ensureIndex( { loc : "2dsphere" , rating : 1 , category : -1 } )
```

The first example creates a simple geospatial index on the location field `loc`. The second example creates a compound index where the second field contains non-location data. The third example creates an index where the location field is not the primary field: the location field does not have to be the first field in a `2dsphere` index. The fourth example creates a compound index with three fields. You can include as many fields as you like in a `2dsphere` index.

Query a 2dsphere Index

The following sections describe queries supported by the `2dsphere` index. For an overview of recommended geospatial queries, see [geospatial-query-compatibility-chart](#).

GeoJSON Objects Bounded by a Polygon

The `$geoWithin` operator queries for location data found within a GeoJSON polygon. Your location data must be stored in GeoJSON format. Use the following syntax:

```
db.<collection>.find( { <location field> :
    { $geoWithin :
        { $geometry :
            { type : "Polygon" ,
              coordinates : [ <coordinates> ]
            } } } } )
```

The following example selects all points and shapes that exist entirely within a GeoJSON polygon:

```
db.places.find( { loc :
    { $geoWithin :
        { $geometry :
            { type : "Polygon" ,
              coordinates : [ [
                  [ [ 0 , 0 ] ,
                    [ 3 , 6 ] ,
                    [ 6 , 1 ] ,
                    [ 0 , 0 ]
                  ] ]
                } } } } )
```

Intersections of GeoJSON Objects

New in version 2.4.

The `$geoIntersects` operator queries for locations that intersect a specified GeoJSON object. A location intersects the object if the intersection is non-empty. This includes documents that have a shared edge.

The `$geoIntersects` operator uses the following syntax:

```
db.<collection>.find( { <location field> :
    { $geoIntersects :
        { $geometry :
            { type : "<GeoJSON object type>" ,
              coordinates : [ <coordinates> ]
            } } } } )
```

The following example uses `$geoIntersects` to select all indexed points and shapes that intersect with the polygon defined by the `coordinates` array.

```
db.places.find( { loc :
    { $geoIntersects :
        { $geometry :
            { type : "Polygon" ,
              coordinates: [ [
                  [ [ 0 , 0 ] ,
                    [ 3 , 6 ] ,
                    [ 6 , 1 ] ,
                    [ 0 , 0 ]
                  ] ]
                } } } } )
```

Proximity to a GeoJSON Point

Proximity queries return the points closest to the defined point and sorts the results by distance. A proximity query on GeoJSON data requires a `2dsphere` index.

To query for proximity to a GeoJSON point, use either the `$near` operator or `geoNear` command. Distance is in meters.

The `$near` uses the following syntax:

```
db.<collection>.find( { <location field> :
    { $near :
        { $geometry :
            { type : "Point" ,
              coordinates : [ <longitude> , <latitude> ] }
            , $maxDistance : <distance in meters>
          } } } )
```

For examples, see `$near`.

The `geoNear` command uses the following syntax:

```
db.runCommand( { geoNear: <collection>, near: [ <x> , <y> ] } )
```

The `geoNear` command offers more options and returns more information than does the `$near` operator. To run the command, see `geoNear`.

Points within a Circle Defined on a Sphere

To select all grid coordinates in a “spherical cap” on a sphere, use `$geoWithin` with the `$centerSphere` operator. Specify an array that contains:

- The grid coordinates of the circle's center point
- The circle's radius measured in radians. To calculate radians, see [Calculate Distance Using Spherical Geometry](#) (page 352).

Use the following syntax:

```
db.<collection>.find( { <location field> :  
    { $geoWithin :  
        { $centerSphere :  
            [ [ <x>, <y> ] , <radius> ] }  
    } } )
```

The following example queries grid coordinates and returns all documents within a 10 mile radius of longitude 88° W and latitude 30° N. The example converts the distance, 10 miles, to radians by dividing by the approximate radius of the earth, 3959 miles:

```
db.places.find( { loc :  
    { $geoWithin :  
        { $centerSphere :  
            [ [ 88 , 30 ] , 10 / 3959 ]  
        } } } )
```

Create a 2d Index

To build a geospatial 2d index, use the `ensureIndex()` method and specify `2d`. Use the following syntax:

```
db.<collection>.ensureIndex( { <location field> : "2d" ,  
                             <additional field> : <value> } ,  
                             { <index-specification options> } )
```

The `2d` index uses the following optional index-specification options:

```
{ min : <lower bound> , max : <upper bound> ,  
  bits : <bit precision> }
```

Define Location Range for a 2d Index

By default, a `2d` index assumes longitude and latitude and has boundaries of -180 inclusive and 180 non-inclusive (i.e. `[-180 , 180]`). If documents contain coordinate data outside of the specified range, MongoDB returns an error.

Important: The default boundaries allow applications to insert documents with invalid latitudes greater than 90 or less than -90. The behavior of geospatial queries with such invalid points is not defined.

On `2d` indexes you can change the location range.

You can build a `2d` geospatial index with a location range other than the default. Use the `min` and `max` options when creating the index. Use the following syntax:

```
db.collection.ensureIndex( { <location field> : "2d" } ,  
                           { min : <lower bound> , max : <upper bound> } )
```

Define Location Precision for a 2d Index

By default, a 2d index on legacy coordinate pairs uses 26 bits of precision, which is roughly equivalent to 2 feet or 60 centimeters of precision using the default range of -180 to 180. Precision is measured by the size in bits of the *geohash* values used to store location data. You can configure geospatial indexes with up to 32 bits of precision.

Index precision does not affect query accuracy. The actual grid coordinates are always used in the final query processing. Advantages to lower precision are a lower processing overhead for insert operations and use of less space. An advantage to higher precision is that queries scan smaller portions of the index to return results.

To configure a location precision other than the default, use the `bits` option when creating the index. Use following syntax:

```
db.<collection>.ensureIndex( {<location field> : "<index type>"} ,  
                           { bits : <bit precision> } )
```

For information on the internals of geohash values, see [Calculation of Geohash Values for 2d Indexes](#) (page 327).

Query a 2d Index

The following sections describe queries supported by the 2d index. For an overview of recommended geospatial queries, see [geospatial-query-compatibility-chart](#).

Points within a Shape Defined on a Flat Surface

To select all legacy coordinate pairs found within a given shape on a flat surface, use the `$geoWithin` operator along with a shape operator. Use the following syntax:

```
db.<collection>.find( { <location field> :  
                        { $geoWithin :  
                            { $box|$polygon|$center : <coordinates>  
                            } } } )
```

The following queries for documents within a rectangle defined by [0 , 0] at the bottom left corner and by [100 , 100] at the top right corner.

```
db.places.find( { loc :  
                  { $geoWithin :  
                      { $box : [ [ 0 , 0 ] ,  
                                [ 100 , 100 ] ]  
                      } } } )
```

The following queries for documents that are within the circle centered on [-74 , 40.74] and with a radius of 10:

```
db.places.find( { loc: { $geoWithin :  
                         { $center : [ [-74, 40.74] , 10 ]  
                         } } } )
```

For syntax and examples for each shape, see the following:

- `$box`
- `$polygon`
- `$center` (defines a circle)

Points within a Circle Defined on a Sphere

MongoDB supports rudimentary spherical queries on flat 2d indexes for legacy reasons. In general, spherical calculations should use a 2dsphere index, as described in [2dsphere Indexes](#) (page 324).

To query for legacy coordinate pairs in a “spherical cap” on a sphere, use \$geoWithin with the \$centerSphere operator. Specify an array that contains:

- The grid coordinates of the circle’s center point
- The circle’s radius measured in radians. To calculate radians, see [Calculate Distance Using Spherical Geometry](#) (page 352).

Use the following syntax:

```
db.<collection>.find( { <location field> :
    { $geoWithin :
        { $centerSphere : [ [ <x>, <y> ] , <radius> ] }
    } } )
```

The following example query returns all documents within a 10-mile radius of longitude 88° W and latitude 30° N. The example converts distance to radians by dividing distance by the approximate radius of the earth, 3959 miles:

```
db.<collection>.find( { loc : { $geoWithin :
    { $centerSphere :
        [ [ 88 , 30 ] , 10 / 3959 ]
    } } } )
```

Proximity to a Point on a Flat Surface

Proximity queries return the 100 legacy coordinate pairs closest to the defined point and sort the results by distance. Use either the \$near operator or geoNear command. Both require a 2d index.

The \$near operator uses the following syntax:

```
db.<collection>.find( { <location field> :
    { $near : [ <x> , <y> ]
    } } )
```

For examples, see \$near.

The geoNear command uses the following syntax:

```
db.runCommand( { geoNear: <collection>, near: [ <x> , <y> ] } )
```

The geoNear command offers more options and returns more information than does the \$near operator. To run the command, see geoNear.

Exact Matches on a Flat Surface

You can use the db.collection.find() method to query for an exact match on a location. These queries use the following syntax:

```
db.<collection>.find( { <location field>: [ <x> , <y> ] } )
```

This query will return any documents with the value of [<x> , <y>].

Create a Haystack Index

To build a haystack index, use the `bucketSize` option when creating the index. A `bucketSize` of 5 creates an index that groups location values that are within 5 units of the specified longitude and latitude. The `bucketSize` also determines the granularity of the index. You can tune the parameter to the distribution of your data so that in general you search only very small regions. The areas defined by buckets can overlap. A document can exist in multiple buckets.

A haystack index can reference two fields: the location field and a second field. The second field is used for exact matches. Haystack indexes return documents based on location and an exact match on a single additional criterion. These indexes are not necessarily suited to returning the closest documents to a particular location.

To build a haystack index, use the following syntax:

```
db.coll.ensureIndex( { <location field> : "geoHaystack" ,
                      <additional field> : 1 } ,
                      { bucketSize : <bucket value> } )
```

Example

If you have a collection with documents that contain fields similar to the following:

```
{ _id : 100, pos: { lng : 126.9, lat : 35.2 } , type : "restaurant" }
{ _id : 200, pos: { lng : 127.5, lat : 36.1 } , type : "restaurant" }
{ _id : 300, pos: { lng : 128.0, lat : 36.7 } , type : "national park" }
```

The following operations create a haystack index with buckets that store keys within 1 unit of longitude or latitude.

```
db.places.ensureIndex( { pos : "geoHaystack" , type : 1 } ,
                      { bucketSize : 1 } )
```

This index stores the document with an `_id` field that has the value 200 in two different buckets:

- In a bucket that includes the document where the `_id` field has a value of 100
 - In a bucket that includes the document where the `_id` field has a value of 300
-

To query using a haystack index you use the `geoSearch` command. See [Query a Haystack Index](#) (page 351).

By default, queries that use a haystack index return 50 documents.

Query a Haystack Index

A haystack index is a special 2d geospatial index that is optimized to return results over small areas. To create a haystack index see [Create a Haystack Index](#) (page 351).

To query a haystack index, use the `geoSearch` command. You must specify both the coordinates and the additional field to `geoSearch`. For example, to return all documents with the value `restaurant` in the `type` field near the example point, the command would resemble:

```
db.runCommand( { geoSearch : "places" ,
                  search : { type: "restaurant" } ,
                  near : [-74, 40.74] ,
                  maxDistance : 10 } )
```

Note: Haystack indexes are not suited to queries for the complete list of documents closest to a particular location. The closest documents could be more distant compared to the bucket size.

Note: *Spherical query operations* (page 352) are not currently supported by haystack indexes.

The `find()` method and `geoNear` command cannot access the haystack index.

Calculate Distance Using Spherical Geometry

Note: While basic queries using spherical distance are supported by the `2d` index, consider moving to a `2dsphere` index if your data is primarily longitude and latitude.

The `2d` index supports queries that calculate distances on a Euclidean plane (flat surface). The index also supports the following query operators and command that calculate distances using spherical geometry:

- `$nearSphere`
- `$centerSphere`
- `$near`
- `geoNear` command with the `{ spherical: true }` option.

Important: These three queries use radians for distance. Other query types do not.

For spherical query operators to function properly, you must convert distances to radians, and convert from radians to the distances units used by your application.

To convert:

- *distance to radians*: divide the distance by the radius of the sphere (e.g. the Earth) in the same units as the distance measurement.
- *radians to distance*: multiply the radian measure by the radius of the sphere (e.g. the Earth) in the units system that you want to convert the distance to.

The radius of the Earth is approximately 3,959 miles or 6,371 kilometers.

The following query would return documents from the `places` collection within the circle described by the center `[-74, 40.74]` with a radius of 100 miles:

```
db.places.find( { loc: { $geoWithin: { $centerSphere: [ [ -74, 40.74 ],  
                                              100 / 3959 ] } } } )
```

You may also use the `distanceMultiplier` option to the `geoNear` to convert radians in the `mongod` process, rather than in your application code. See *distance multiplier* (page 353).

The following spherical query, returns all documents in the collection `places` within 100 miles from the point `[-74, 40.74]`.

```
db.runCommand( { geoNear: "places",  
                near: [ -74, 40.74 ],  
                spherical: true  
              } )
```

The output of the above command would be:

```
{  
  // [...]  
  "results" : [  
    {  
      "dis" : 0.01853688938212826,  
      "obj" : {
```

```

        "_id" : ObjectId( ... )
        "loc" : [
            -73,
            40
        ]
    }
],
"stats" : {
    // [ ... ]
    "avgDistance" : 0.01853688938212826,
    "maxDistance" : 0.01853714811400047
},
"ok" : 1
}

```

Warning: Spherical queries that wrap around the poles or at the transition from -180 to 180 longitude raise an error.

Note: While the default Earth-like bounds for geospatial indexes are between -180 inclusive, and 180 , valid values for latitude are between -90 and 90 .

Distance Multiplier

The `distanceMultiplier` option of the `geoNear` command returns distances only after multiplying the results by an assigned value. This allows MongoDB to return converted values, and removes the requirement to convert units in application logic.

Using `distanceMultiplier` in spherical queries provides results from the `geoNear` command that do not need radian-to-distance conversion. The following example uses `distanceMultiplier` in the `geoNear` command with a [spherical](#) (page 352) example:

```
db.runCommand( { geoNear: "places",
                 near: [ -74, 40.74 ],
                 spherical: true,
                 distanceMultiplier: 3959
               } )
```

The output of the above operation would resemble the following:

```
{
    // [ ... ]
    "results" : [
        {
            "dis" : 73.46525170413567,
            "obj" : {
                "_id" : ObjectId( ... )
                "loc" : [
                    -73,
                    40
                ]
            }
        }
    ],
    "stats" : {
```

```
// [ ... ]
"avgDistance" : 0.01853688938212826,
"maxDistance" : 0.01853714811400047
},
"ok" : 1
}
```

7.3.4 Text Search Tutorials

Instructions for enabling MongoDB's text search feature, and for building and configuring text indexes.

[Enable Text Search \(page 354\)](#) You must explicitly enable text search in order to search string content in collections.

[Create a text Index \(page 355\)](#) A `text` index allows searches on text strings in the index's specified fields.

[Search String Content for Text \(page 355\)](#) Use queries to find strings of text within collections.

[Specify a Language for Text Index \(page 358\)](#) The specified language determines the list of stop words and the rules for Text Search's stemmer and tokenizer.

[Create text Index with Long Name \(page 360\)](#) Override the `text` index name limit for long index names.

[Control Search Results with Weights \(page 360\)](#) Give priority to certain search values by denoting the significance of an indexed field relative to other indexed fields

[Limit the Number of Entries Scanned \(page 361\)](#) Search only those documents that match a set of filter conditions.

[Create text Index to Cover Queries \(page 363\)](#) Perform text searches that return results without the need to scan documents.

Enable Text Search

New in version 2.4.

The `text search` (page 329) is currently a *beta* feature. As a beta feature:

- You need to explicitly enable the feature before [creating a text index \(page 328\)](#) or using the `text` command.
- To enable text search on [replica sets \(page 377\)](#) and [sharded clusters \(page 492\)](#), you need to enable on **each and every** `mongod` for replica sets and on **each and every** `mongos` for sharded clusters.

Warning:

- Do **not** enable or use text search on production systems.
- Text indexes have significant storage requirements and performance costs. See [Storage Requirements and Performance Costs \(page 328\)](#) for more information.

You can enable the text search feature at startup with the `textSearchEnabled` parameter:

```
mongod --setParameter textSearchEnabled=true
```

You may prefer to set the `textSearchEnabled` parameter in the configuration file.

Additionally, you can enable the feature in the `mongo` shell with the `setParameter` command. This command does **not** propagate from the primary to the secondaries. You must enable on **each and every** `mongod` for replica sets.

Note: You must set the parameter every time you start the server. You may prefer to add the parameter to the configuration files.

Create a `text` Index

You can create a `text` index on the field or fields whose value is a string or an array of string elements. When creating a `text` index on multiple fields, you can specify the individual fields or you can wildcard specifier (`$**`).

Index Specific Fields

The following example creates a `text` index on the fields `subject` and `content`:

```
db.collection.ensureIndex(
  {
    subject: "text",
    content: "text"
  }
)
```

This `text` index catalogs all string data in the `subject` field and the `content` field, where the field value is either a string or an array of string elements.

Index All Fields

To allow for text search on all fields with string content, use the wildcard specifier (`$**`) to index all fields that contain string content.

The following example indexes any string value in the data of every field of every document in `collection` and names the index `TextIndex`:

```
db.collection.ensureIndex(
  { "$**": "text" },
  { name: "TextIndex" }
)
```

Search String Content for Text

In 2.4, you can enable the text search feature to create `text` indexes and issue text queries using the `text`.

The following tutorial offers various query patterns for using the text search feature.

The examples in this tutorial use a collection `quotes` that has a `text` index on the fields `quote` that contains a string and `related_quotes` that contains an array of string elements.

Note: You cannot combine the `text` command, which requires a special [text index](#) (page 328), with a query operator that requires a different type of special index. For example you cannot combine `text` with the `$near` operator.

Search for a Term

The following command searches for the word `TOMORROW`:

```
db.quotes.runCommand( "text", { search: "TOMORROW" } )
```

Because `text` command is case-insensitive, the text search will match the following document in the `quotes` collection:

```
{  
    "_id" : ObjectId("50ecef5f8abea0fda30ceab3"),  
    "quote" : "tomorrow, and tomorrow, and tomorrow, creeps in this petty pace",  
    "related_quotes" : [  
        "is this a dagger which I see before me",  
        "the handle toward my hand?"  
    ],  
    "src" : {  
        "title" : "Macbeth",  
        "from" : "Act V, Scene V"  
    },  
    "speaker" : "macbeth"  
}
```

Match Any of the Search Terms

If the search string is a space-delimited text, `text` command performs a logical OR search on each term and returns documents that contains any of the terms.

For example, the search string "tomorrow largo" searches for the term tomorrow **OR** the term largo:

```
db.quotes.runCommand( "text", { search: "tomorrow largo" } )
```

The command will match the following documents in the quotes collection:

```
{  
    "_id" : ObjectId("50ecef5f8abea0fda30ceab3"),  
    "quote" : "tomorrow, and tomorrow, and tomorrow, creeps in this petty pace",  
    "related_quotes" : [  
        "is this a dagger which I see before me",  
        "the handle toward my hand?"  
    ],  
    "src" : {  
        "title" : "Macbeth",  
        "from" : "Act V, Scene V"  
    },  
    "speaker" : "macbeth"  
}  
  
{  
    "_id" : ObjectId("50ecf0cd8abea0fda30ceab4"),  
    "quote" : "Es tan corto el amor y es tan largo el olvido.",  
    "related_quotes" : [  
        "Como para acercarla mi mirada la busca.",  
        "Mi corazón la busca, y ella no está conmigo."  
    ],  
    "speaker" : "Pablo Neruda",  
    "src" : {  
        "title" : "Veinte poemas de amor y una canción desesperada",  
        "from" : "Poema 20"  
    }  
}
```

Match Phrases

To match the exact phrase that includes a space(s) as a single term, escape the quotes.

For example, the following command searches for the exact phrase "and tomorrow":

```
db.quotes.runCommand( "text", { search: "\"and tomorrow\"" } )
```

If the search string contains both phrases and individual terms, the `text` command performs a compound logical AND of the phrases with the compound logical OR of the single terms, including the individual terms from each phrase.

For example, the following search string contains both individual terms `corto` and `largo` as well as the phrase `\"and tomorrow\"`:

```
db.quotes.runCommand( "text", { search: "corto largo \"and tomorrow\"" } )
```

The `text` command performs the equivalent to the following logical operation, where the individual terms `corto`, `largo`, as well as the term `tomorrow` from the phrase "and tomorrow", are part of a logical OR expression:

```
(corto OR largo OR tomorrow) AND ("and tomorrow")
```

As such, the results for this search will include documents that only contain the phrase "and tomorrow" as well as documents that contain the phrase "and tomorrow" and the terms `corto` and/or `largo`. Documents that contain the phrase "and tomorrow" as well as the terms `corto` and `largo` will generally receive a higher score for this search.

Match Some Words But Not Others

A *negated* term is a term that is prefixed by a minus sign `-`. If you negate a term, the `text` command will exclude the documents that contain those terms from the results.

Note: If the search text contains *only* negated terms, the `text` command will not return any results.

The following example returns those documents that contain the term `tomorrow` but **not** the term `petty`.

```
db.quotes.runCommand( "text" , { search: "tomorrow -petty" } )
```

Limit the Number of Matching Documents in the Result Set

Note: The result from the `text` command must fit within the maximum BSON Document Size.

By default, the `text` command will return up to 100 matching documents, from highest to lowest scores. To override this default limit, use the `limit` option in the `text` command, as in the following example:

```
db.quotes.runCommand( "text" , { search: "tomorrow" , limit: 2 } )
```

The `text` command will return at most 2 of the *highest scoring* results.

The `limit` can be any number as long as the result set fits within the maximum BSON Document Size.

Specify Which Fields to Return in the Result Set

In the `text` command, use the `project` option to specify the fields to include (1) or exclude (0) in the matching documents.

Note: The `_id` field is always returned unless explicitly excluded in the `project` document.

The following example returns only the `_id` field and the `src` field in the matching documents:

```
db.quotes.runCommand( "text", { search: "tomorrow",
                                project: { "src": 1 } } )
```

Search with Additional Query Conditions

The `text` command can also use the `filter` option to specify additional query conditions.

The following example will return the documents that contain the term `tomorrow` **AND** the speaker is `macbeth`:

```
db.quotes.runCommand( "text", { search: "tomorrow",
                                filter: { speaker : "macbeth" } } )
```

See also:

[Limit the Number of Entries Scanned](#) (page 361)

Search for Text in Specific Languages

You can specify the language that determines the tokenization, stemming, and removal of stop words, as in the following example:

```
db.quotes.runCommand( "text", { search: "amor", language: "spanish" } )
```

See `text-search-languages` for a list of supported languages as well as [Specify a Language for Text Index](#) (page 358) for specifying languages for the `text` index.

Text Search Output

The `text` command returns a document that contains the result set.

See `text-search-output` for information on the output.

Specify a Language for Text Index

This tutorial describes how to [specify the default language associated with the text index](#) (page 358) and also how to [create text indexes for collections that contain documents in different languages](#) (page 359).

Specify the Default Language for a `text` Index

The default language associated with the indexed data determines the list of stop words and the rules for the stemmer and tokenizer. The default language for the indexed data is `english`.

To specify a different language, use the `default_language` option when creating the `text` index. See `text-search-languages` for the languages available for `default_language`.

The following example creates a `text` index on the `content` field and sets the `default_language` to `spanish`:

```
db.collection.ensureIndex(
    { content : "text" },
    { default_language: "spanish" }
)
```

Create a `text` Index for a Collection in Multiple Languages

Specify the Index Language within the Document If a collection contains documents that are in different languages, include a field in the documents that contain the language to use:

- If you include a field named `language` in the document, by default, the `ensureIndex()` method will use the value of this field to override the default language.
- To use a field with a name other than `language`, you must specify the name of this field to the `ensureIndex()` method with the `language_override` option.

See `text-search-languages` for a list of supported languages.

Include the `language` Field Include a field `language` that specifies the language to use for the individual documents.

For example, the documents of a multi-language collection `quotes` contain the field `language`:

```
{ _id: 1, language: "portuguese", quote: "A sorte protege os audazes" }
{ _id: 2, language: "spanish", quote: "Nada hay más surreal que la realidad." }
{ _id: 3, language: "english", quote: "is this a dagger which I see before me" }
```

Create a `text` index on the field `quote`:

```
db.quotes.ensureIndex( { quote: "text" } )
```

- For the documents that contain the `language` field, the `text` index uses that language to determine the stop words and the rules for the stemmer and the tokenizer.
- For documents that do not contain the `language` field, the index uses the default language, which is English, to determine the stop words and rules for the stemmer and the tokenizer.

For example, the Spanish word `que` is a stop word. So the following `text` command would not match any document:

```
db.quotes.runCommand( "text", { search: "que", language: "spanish" } )
```

Use any Field to Specify the Language for a Document Include a field that specifies the language to use for the individual documents. To use a field with a name other than `language`, include the `language_override` option when creating the index.

For example, the documents of a multi-language collection `quotes` contain the field `idioma`:

```
{ _id: 1, idioma: "portuguese", quote: "A sorte protege os audazes" }
{ _id: 2, idioma: "spanish", quote: "Nada hay más surreal que la realidad." }
{ _id: 3, idioma: "english", quote: "is this a dagger which I see before me" }
```

Create a `text` index on the field `quote` with the `language_override` option:

```
db.quotes.ensureIndex( { quote : "text" },
                      { language_override: "idioma" } )
```

- For the documents that contain the `idioma` field, the `text` index uses that language to determine the stop words and the rules for the stemmer and the tokenizer.
- For documents that do not contain the `idioma` field, the index uses the default language, which is English, to determine the stop words and rules for the stemmer and the tokenizer.

For example, the Spanish word `que` is a stop word. So the following `text` command would not match any document:

```
db.quotes.runCommand( "text", { search: "que", language: "spanish" } )
```

Create text Index with Long Name

The default name for the index consists of each indexed field name concatenated with `_text`. For example, the following command creates a `text` index on the fields `content`, `users.comments`, and `users.profiles`:

```
db.collection.ensureIndex(
  {
    content: "text",
    "users.comments": "text",
    "users.profiles": "text"
  }
)
```

The default name for the index is:

```
"content_text_users.comments_text_users.profiles_text"
```

To avoid creating an index with a name that exceeds the index name length limit, you can pass the `name` option to the `db.collection.ensureIndex()` method:

```
db.collection.ensureIndex(
  {
    content: "text",
    "users.comments": "text",
    "users.profiles": "text"
  },
  {
    name: "MyTextIndex"
  }
)
```

Note: To drop the `text` index, use the index name. To get the name of an index, use `db.collection.getIndexes()`.

Control Search Results with Weights

This document describes how to create a `text` index with specified weights for results fields.

By default, the `text` command returns matching documents based on scores, from highest to lowest. For a `text` index, the *weight* of an indexed field denotes the significance of the field relative to the other indexed fields in terms of the score. The score for a given word in a document is derived from the weighted sum of the frequency for each of the indexed fields in that document.

The default weight is 1 for the indexed fields. To adjust the weights for the indexed fields, include the `weights` option in the `db.collection.ensureIndex()` method.

Warning: Choose the weights carefully in order to prevent the need to reindex.

A collection `blog` has the following documents:

```
{ _id: 1,
  content: "This morning I had a cup of coffee.",
  about: "beverage",
```

```

    keywords: [ "coffee" ]
}

{ _id: 2,
  content: "Who doesn't like cake?",
  about: "food",
  keywords: [ "cake", "food", "dessert" ]
}

```

To create a `text` index with different field weights for the `content` field and the `keywords` field, include the `weights` option to the `ensureIndex()` method. For example, the following command creates an index on three fields and assigns weights to two of the fields:

```

db.blog.ensureIndex(
  {
    content: "text",
    keywords: "text",
    about: "text"
  },
  {
    weights: {
      content: 10,
      keywords: 5,
    },
    name: "TextIndex"
  }
)

```

The `text` index has the following fields and weights:

- `content` has a weight of 10,
- `keywords` has a weight of 5, and
- `about` has the default weight of 1.

These weights denote the relative significance of the indexed fields to each other. For instance, a term match in the `content` field has:

- 2 times (i.e. 10 : 5) the impact as a term match in the `keywords` field and
- 10 times (i.e. 10 : 1) the impact as a term match in the `about` field.

Limit the Number of Entries Scanned

This tutorial describes how to limit the text search to scan only those documents with a field value.

The `text` command includes the `filter` option to further restrict the results of a text search. For a `filter` that specifies equality conditions, this tutorial demonstrates how to perform text searches on only those documents that match the `filter` conditions, as opposed to performing a text search first on all the documents and then matching on the `filter` condition.

Consider a collection `inventory` that contains the following documents:

```

{ _id: 1, dept: "tech", description: "a fun green computer" }
{ _id: 2, dept: "tech", description: "a wireless red mouse" }
{ _id: 3, dept: "kitchen", description: "a green placemat" }
{ _id: 4, dept: "kitchen", description: "a red peeler" }
{ _id: 5, dept: "food", description: "a green apple" }
{ _id: 6, dept: "food", description: "a red potato" }

```

A common use case is to perform text searches by individual departments, such as:

```
db.inventory.runCommand( "text", {  
    search: "green",  
    filter: { dept : "kitchen" }  
}  
)
```

To limit the text search to scan only those documents within a specific dept, create a compound index that specifies an ascending/descending index key on the field dept and a text index key on the field description:

```
db.inventory.ensureIndex(  
    {  
        dept: 1,  
        description: "text"  
    }  
)
```

Important:

- The ascending/descending index keys must be listed before, or prefix, the text index keys.
 - By prefixing the text index fields with ascending/descending index fields, MongoDB will **only** index documents that have the prefix fields.
 - You cannot include *multi-key* (page 320) index fields or *geospatial* (page 323) index fields.
 - The text command **must** include the filter option that specifies an **equality** condition for the prefix fields.
-

Then, the text search within a particular department will limit the scan of indexed documents. For example, the following text command scans only those documents with dept equal to kitchen:

```
db.inventory.runCommand( "text", {  
    search: "green",  
    filter: { dept : "kitchen" }  
}  
)
```

The returned result includes the statistics that shows that the command scanned 1 document, as indicated by the nscanned field:

```
{  
  
    "queryDebugString" : "green|||||",  
    "language" : "english",  
    "results" : [  
        {  
            "score" : 0.75,  
            "obj" : {  
                "_id" : 3,  
                "dept" : "kitchen",  
                "description" : "a green placemat"  
            }  
        }  
    ],  
    "stats" : {  
        "nscanned" : 1,  
        "nscannedObjects" : 0,  
        "n" : 1,  
        "nfound" : 1,
```

```

        "timeMicros" : 211
    },
    "ok" : 1
}

```

For more information on the result set, see [text-search-output](#).

Create text Index to Cover Queries

To create a `text` index that can [cover queries](#) (page 365):

1. Append scalar index fields to a `text` index, as in the following example which specifies an ascending index key on `username`:

```
db.collection.ensureIndex( { comments: "text",
                            username: 1 } )
```

Warning: You cannot include [multi-key](#) (page 320) index field or [geospatial](#) (page 323) index field.

2. Use the `project` option in the `text` to return only the fields in the index, as in the following:

```
db.quotes.runCommand( "text", { search: "tomorrow",
                               project: { username: 1,
                                          _id: 0
                                         }
                             }
                       )
```

Note: By default, the `_id` field is included in the result set. Since the example index did not include the `_id` field, you must explicitly exclude the field in the `project` document.

7.3.5 Indexing Strategies

The best indexes for your application must take a number of factors into account, including the kinds of queries you expect, the ratio of reads to writes, and the amount of free memory on your system.

When developing your indexing strategy you should have a deep understanding of your application's queries. Before you build indexes, map out the types of queries you will run so that you can build indexes that reference those fields. Indexes come with a performance cost, but are more than worth the cost for frequent queries on large data set. Consider the relative frequency of each query in the application and whether the query justifies an index.

The best overall strategy for designing indexes is to profile a variety of index configurations with data sets similar to the ones you'll be running in production to see which configurations perform best. Inspect the current indexes created for your collections to ensure they are supporting your current and planned queries. If an index is no longer used, drop the index.

MongoDB can only use *one* index to support any given operation. However, each clause of an `$or` query may use a different index.

The following documents introduce indexing strategies:

[Create Indexes to Support Your Queries \(page 364\)](#) An index supports a query when the index contains all the fields scanned by the query. Creating indexes that supports queries results in greatly increased query performance.

[Use Indexes to Sort Query Results \(page 366\)](#) To support efficient queries, use the strategies here when you specify the sequential order and sort order of index fields.

[Ensure Indexes Fit in RAM](#) (page 368) When your index fits in RAM, the system can avoid reading the index from disk and you get the fastest processing.

[Create Queries that Ensure Selectivity](#) (page 368) Selectivity is the ability of a query to narrow results using the index. Selectivity allows MongoDB to use the index for a larger portion of the work associated with fulfilling the query.

Create Indexes to Support Your Queries

An index supports a query when the index contains all the fields scanned by the query. The query scans the index and not the collection. Creating indexes that supports queries results in greatly increased query performance.

This document describes strategies for creating indexes that support queries.

Create a Single-Key Index if All Queries Use the Same, Single Key

If you only ever query on a single key in a given collection, then you need to create just one single-key index for that collection. For example, you might create an index on `category` in the `product` collection:

```
db.products.ensureIndex( { "category": 1 } )
```

Create Compound Indexes to Support Several Different Queries

If you sometimes query on only one key and at other times query on that key combined with a second key, then creating a compound index is more efficient than creating a single-key index. MongoDB will use the compound index for both queries. For example, you might create an index on both `category` and `item`.

```
db.products.ensureIndex( { "category": 1, "item": 1 } )
```

This allows you both options. You can query on just `category`, and you also can query on `category` combined with `item`. A single [compound index](#) (page 318) on multiple fields can support all the queries that search a “prefix” subset of those fields.

Note: With the exception of queries that use the `$or` operator, a query does not use multiple indexes. A query uses only one index.

Example

The following index on a collection:

```
{ x: 1, y: 1, z: 1 }
```

Can support queries that the following indexes support:

```
{ x: 1 }
{ x: 1, y: 1 }
```

There are some situations where the prefix indexes may offer better query performance: for example if `z` is a large array.

The `{ x: 1, y: 1, z: 1 }` index can also support many of the same queries as the following index:

```
{ x: 1, z: 1 }
```

Also, `{ x: 1, z: 1 }` has an additional use. Given the following query:

```
db.collection.find( { x: 5 } ).sort( { z: 1 } )
```

The `{ x: 1, z: 1 }` index supports both the query and the sort operation, while the `{ x: 1, y: 1, z: 1 }` index only supports the query. For more information on sorting, see [Use Indexes to Sort Query Results](#) (page 366).

Create Indexes that Support Covered Queries

A covered query is a query in which:

- all the fields in the [query](#) (page 57) are part of an index, **and**
- all the fields returned in the results are in the same index.

Because the index “covers” the query, MongoDB can both match the [query conditions](#) (page 57) **and** return the results using only the index; MongoDB does not need to look at the documents, only the index, to fulfill the query. An index can also cover an [aggregation pipeline operation](#) (page 279) on unsharded collections.

Querying *only* the index can be much faster than querying documents outside of the index. Index keys are typically smaller than the documents they catalog, and indexes are typically available in RAM or located sequentially on disk.

MongoDB automatically uses an index that covers a query when possible. To ensure that an index can *cover* a query, create an index that includes all the fields listed in the [query document](#) (page 57) and in the query result. You can specify the fields to return in the query results with a [projection](#) (page 61) document. By default, MongoDB includes the `_id` field in the query result. So, if the index does **not** include the `_id` field, then you must exclude the `_id` field (i.e. `_id: 0`) from the query results.

Example

Given collection `users` with an index on the fields `status` and `user`, as created by the following option:

```
db.users.ensureIndex( { status: 1, user: 1 } )
```

Then, this index will cover the following query which selects on the `status` field and returns only the `user` field:

```
db.users.find( { status: "A" }, { user: 1, _id: 0 } )
```

In the operation, the projection document explicitly specifies `_id: 0` to exclude the `_id` field from the result since the index is only on the `status` and the `user` fields.

If the projection document does not specify the exclusion of the `_id` field, the query returns the `_id` field. The following query is **not** covered by the index on the `status` and the `user` fields because with the projection document `{ user: 1 }`, the query returns both the `user` field and the `_id` field:

```
db.users.find( { status: "A" }, { user: 1 } )
```

An index **cannot** cover a query if:

- any of the indexed fields in any of the documents in the collection includes an array. If an indexed field is an array, the index becomes a [multi-key index](#) (page 320) index and cannot support a covered query.
- any of the indexed fields are fields in subdocuments. To index fields in subdocuments, use *dot notation*. For example, consider a collection `users` with documents of the following form:

```
{ _id: 1, user: { login: "tester" } }
```

The collection has the following indexes:

```
{ user: 1 }  
{ "user.login": 1 }
```

The { user: 1 } index covers the following query:

```
db.users.find( { user: { login: "tester" } }, { user: 1, _id: 0 } )
```

However, the { "user.login": 1 } index does **not** cover the following query:

```
db.users.find( { "user.login": "tester" }, { "user.login": 1, _id: 0 } )
```

The query, however, does use the { "user.login": 1 } index to find matching documents.

To determine whether a query is a covered query, use the `explain()` method. If the `explain()` output displays `true` for the `indexOnly` field, the query is covered by an index, and MongoDB queries only that index to match the query **and** return the results.

For more information see [Measure Index Use](#) (page 344).

Use Indexes to Sort Query Results

In MongoDB sort operations that sort documents based on an indexed field provide the greatest performance. Indexes in MongoDB, as in other databases, have an order: as a result, using an index to access documents returns in the same order as the index.

To sort on multiple fields, create a [compound index](#) (page 318). With compound indexes, the results can be in the sorted order of either the full index or an index prefix. An index prefix is a subset of a compound index; the subset consists of one or more fields at the start of the index, in order. For example, given an index { a:1, b: 1, c: 1, d: 1 }, the following subsets are index prefixes:

```
{ a: 1 }  
{ a: 1, b: 1 }  
{ a: 1, b: 1, c: 1 }
```

For more information on sorting by index prefixes, see [Sort Subset Starts at the Index Beginning](#) (page 367).

If the query includes **equality** match conditions on an index prefix, you can sort on a subset of the index that starts after or overlaps with the prefix. For example, given an index { a: 1, b: 1, c: 1, d: 1 }, if the query condition includes equality match conditions on a and b, you can specify a sort on the subsets { c: 1 } or { c: 1, d: 1 }:

```
db.collection.find( { a: 5, b: 3 } ).sort( { c: 1 } )  
db.collection.find( { a: 5, b: 3 } ).sort( { c: 1, d: 1 } )
```

In these operations, the equality match and the sort documents together cover the index prefixes { a: 1, b: 1, c: 1 } and { a: 1, b: 1, c: 1, d: 1 } respectively.

You can also specify a sort order that includes the prefix; however, since the query condition specifies equality matches on these fields, they are constant in the resulting documents and do not contribute to the sort order:

```
db.collection.find( { a: 5, b: 3 } ).sort( { a: 1, b: 1, c: 1 } )  
db.collection.find( { a: 5, b: 3 } ).sort( { a: 1, b: 1, c: 1, d: 1 } )
```

For more information on sorting by index subsets that are not prefixes, see [Sort Subset Does Not Start at the Index Beginning](#) (page 367).

Note: For in-memory sorts that do not use an index, the `sort()` operation is significantly slower. The `sort()` operation will abort when it uses 32 megabytes of memory.

Sort With a Subset of Compound Index

If the sort document contains a subset of the compound index fields, the subset can determine whether MongoDB can use the index efficiently to both retrieve and sort the query results. If MongoDB can efficiently use the index to both retrieve and sort the query results, the output from the `explain()` will display `scanAndOrder` as `false` or `0`. If MongoDB can only use the index for retrieving documents that meet the query criteria, MongoDB must manually sort the resulting documents without the use of the index. For in-memory sort operations, `explain()` will display `scanAndOrder` as `true` or `1`.

Sort Subset Starts at the Index Beginning If the sort document is a subset of a compound index and starts from the beginning of the index, MongoDB can use the index to both retrieve and sort the query results.

For example, the collection `collection` has the following index:

```
{ a: 1, b: 1, c: 1, d: 1 }
```

The following operations include a sort with a subset of the index. Because the sort subset starts at beginning of the index, the operations can use the index for both the query retrieval and sort:

```
db.collection.find().sort( { a:1 } )
db.collection.find().sort( { a:1, b:1 } )
db.collection.find().sort( { a:1, b:1, c:1 } )

db.collection.find( { a: 4 } ).sort( { a: 1, b: 1 } )
db.collection.find( { a: { $gt: 4 } } ).sort( { a: 1, b: 1 } )

db.collection.find( { b: 5 } ).sort( { a: 1, b: 1 } )
db.collection.find( { b: { $gt:5 }, c: { $gt: 1 } } ).sort( { a: 1, b: 1 } )
```

The last two operations include query conditions on the field `b` but does not include a query condition on the field `a`:

```
db.collection.find( { b: 5 } ).sort( { a: 1, b: 1 } )
db.collection.find( { b: { $gt:5 }, c: { $gt: 1 } } ).sort( { a: 1, b: 1 } )
```

Consider the case where the collection has the index `{ b: 1 }` in addition to the `{ a: 1, b: 1, c: 1, d: 1 }` index. Because of the query condition on `b`, it is not immediately obvious which index MongoDB may select as the “best” index. To explicitly specify the index to use, see `hint()`.

Sort Subset Does Not Start at the Index Beginning The sort document can be a subset of a compound index that does **not** start from the beginning of the index. For instance, `{ c: 1 }` is a subset of the index `{ a: 1, b: 1, c: 1, d: 1 }` that omits the preceding index fields `a` and `b`. MongoDB can use the index efficiently **if** the query document includes all the preceding fields of the index, in this case `a` and `b`, in **equality** conditions. In other words, the equality conditions in the query document and the subset in the sort document **contiguously** cover a prefix of the index.

For example, the collection `collection` has the following index:

```
{ a: 1, b: 1, c: 1, d: 1 }
```

Then following operations can use the index efficiently:

```
db.collection.find( { a: 5 } ).sort( { b: 1, c: 1 } )
db.collection.find( { a: 5, c: 4, b: 3 } ).sort( { d: 1 } )
```

- In the first operation, the query document `{ a: 5 }` with the sort document `{ b: 1, c: 1 }` cover the prefix `{ a:1, b: 1, c: 1 }` of the index.

- In the second operation, the query document { a: 5, c: 4, b: 3 } with the sort document { d: 1 } covers the full index.

Only the index fields preceding the sort subset must have the equality conditions in the query document. The other index fields may have other conditions. The following operations can efficiently use the index since the equality conditions in the query document and the subset in the sort document **contiguously** cover a prefix of the index:

```
db.collection.find( { a: 5, b: 3 } ).sort( { c: 1 } )
db.collection.find( { a: 5, b: 3, c: { $lt: 4 } } ).sort( { c: 1 } )
```

The following operations specify a sort document of { c: 1 }, but the query documents do not contain equality matches on the **preceding** index fields a and b:

```
db.collection.find( { a: { $gt: 2 } } ).sort( { c: 1 } )
db.collection.find( { c: 5 } ).sort( { c: 1 } )
```

These operations **will not** efficiently use the index { a: 1, b: 1, c: 1, d: 1 } and may not even use the index to retrieve the documents.

Ensure Indexes Fit in RAM

For the fastest processing, ensure that your indexes fit entirely in RAM so that the system can avoid reading the index from disk.

To check the size of your indexes, use the `db.collection.totalIndexSize()` helper, which returns data in bytes:

```
> db.collection.totalIndexSize()
4294976499
```

The above example shows an index size of almost 4.3 gigabytes. To ensure this index fits in RAM, you must not only have more than that much RAM available but also must have RAM available for the rest of the *working set*. Also remember:

If you have and use multiple collections, you must consider the size of all indexes on all collections. The indexes and the working set must be able to fit in memory at the same time.

There are some limited cases where indexes do not need to fit in memory. See [Indexes that Hold Only Recent Values in RAM](#) (page 368).

See also:

`collStats` and `db.collection.stats()`

Indexes that Hold Only Recent Values in RAM

Indexes do not have to fit *entirely* into RAM in all cases. If the value of the indexed field increments with every insert, and most queries select recently added documents; then MongoDB only needs to keep the parts of the index that hold the most recent or “right-most” values in RAM. This allows for efficient index use for read and write operations and minimize the amount of RAM required to support the index.

Create Queries that Ensure Selectivity

Selectivity is the ability of a query to narrow results using the index. Effective indexes are more selective and allow MongoDB to use the index for a larger portion of the work associated with fulfilling the query.

To ensure selectivity, write queries that limit the number of possible documents with the indexed field. Write queries that are appropriately selective relative to your indexed data.

Example

Suppose you have a field called `status` where the possible values are `new` and `processed`. If you add an index on `status` you've created a low-selectivity index. The index will be of little help in locating records.

A better strategy, depending on your queries, would be to create a [compound index](#) (page 318) that includes the low-selectivity field and another field. For example, you could create a compound index on `status` and `created_at`.

Another option, again depending on your use case, might be to use separate collections, one for each status.

Example

Consider an index `{ a : 1 }` (i.e. an index on the key `a` sorted in ascending order) on a collection where `a` has three values evenly distributed across the collection:

```
{ _id: ObjectId(), a: 1, b: "ab" }
{ _id: ObjectId(), a: 1, b: "cd" }
{ _id: ObjectId(), a: 1, b: "ef" }
{ _id: ObjectId(), a: 2, b: "jk" }
{ _id: ObjectId(), a: 2, b: "lm" }
{ _id: ObjectId(), a: 2, b: "no" }
{ _id: ObjectId(), a: 3, b: "pq" }
{ _id: ObjectId(), a: 3, b: "rs" }
{ _id: ObjectId(), a: 3, b: "tv" }
```

If you query for `{ a: 2, b: "no" }` MongoDB must scan 3 *documents* in the collection to return the one matching result. Similarly, a query for `{ a: { $gt: 1 }, b: "tv" }` must scan 6 documents, also to return one result.

Consider the same index on a collection where `a` has *nine* values evenly distributed across the collection:

```
{ _id: ObjectId(), a: 1, b: "ab" }
{ _id: ObjectId(), a: 2, b: "cd" }
{ _id: ObjectId(), a: 3, b: "ef" }
{ _id: ObjectId(), a: 4, b: "jk" }
{ _id: ObjectId(), a: 5, b: "lm" }
{ _id: ObjectId(), a: 6, b: "no" }
{ _id: ObjectId(), a: 7, b: "pq" }
{ _id: ObjectId(), a: 8, b: "rs" }
{ _id: ObjectId(), a: 9, b: "tv" }
```

If you query for `{ a: 2, b: "cd" }`, MongoDB must scan only one document to fulfill the query. The index and query are more selective because the values of `a` are evenly distributed *and* the query can select a specific document using the index.

However, although the index on `a` is more selective, a query such as `{ a: { $gt: 5 }, b: "tv" }` would still need to scan 4 documents.

If overall selectivity is low, and if MongoDB must read a number of documents to return results, then some queries may perform faster without indexes. To determine performance, see [Measure Index Use](#) (page 344).

For a conceptual introduction to indexes in MongoDB see [Index Concepts](#) (page 314).

7.4 Indexing Reference

7.4.1 Indexing Methods in the mongo Shell

Name	Description
<code>db.collection.createIndex()</code>	Builds an index on a collection. Use <code>db.collection.ensureIndex()</code> .
<code>db.collection.dropIndex()</code>	Removes a specified index on a collection.
<code>db.collection.dropIndexes()</code>	Removes all indexes on a collection.
<code>db.collection.ensureIndex()</code>	Creates an index if it does not currently exist. If the index exists <code>ensureIndex()</code> does nothing.
<code>db.collection.getIndexes()</code>	Returns an array of documents that describe the existing indexes on a collection.
<code>db.collection.getIndexStats()</code>	Rendered a human-readable view of the data collected by <code>indexStats</code> which reflects B-tree utilization.
<code>db.collection.indexStats()</code>	Rendered a human-readable view of the data collected by <code>indexStats</code> which reflects B-tree utilization.
<code>db.collection.reIndex()</code>	Rebuilds all existing indexes on a collection.
<code>db.collection.totalIndexSize()</code>	Reports the total size used by the indexes on a collection. Provides a wrapper around the <code>totalIndexSize</code> field of the <code>collStats</code> output.
<code>cursor.explain()</code>	Reports on the query execution plan, including index use, for a cursor.
<code>cursor_hint()</code>	Forces MongoDB to use a specific index for a query.
<code>cursor.max()</code>	Specifies an exclusive upper index bound for a cursor. For use with <code>cursor_hint()</code>
<code>cursor.min()</code>	Specifies an inclusive lower index bound for a cursor. For use with <code>cursor_hint()</code>
<code>cursor.snapshot()</code>	Forces the cursor to use the index on the <code>_id</code> field. Ensures that the cursor returns each document, with regards to the value of the <code>_id</code> field, only once.

7.4.2 Indexing Database Commands

Name	Description
<code>dropIndexes</code>	Removes indexes from a collection.
<code>compact</code>	Defragments a collection and rebuilds the indexes.
<code>reIndex</code>	Rebuilds all indexes on a collection.
<code>validate</code>	Internal command that scans for a collection's data and indexes for correctness.
<code>indexStats</code>	Experimental command that collects and aggregates statistics on all indexes.
<code>geoNear</code>	Performs a geospatial query that returns the documents closest to a given point.
<code>geoSearch</code>	Performs a geospatial query that uses MongoDB's <i>haystack index</i> functionality.
<code>geoWalk</code>	An internal command to support geospatial queries.
<code>checkShardingIndex</code>	Internal command that validates index on shard key.

7.4.3 Geospatial Query Selectors

Name	Description
<code>\$geoWithin</code>	Selects geometries within a bounding <i>GeoJSON</i> geometry.
<code>\$geoIntersects</code>	Selects geometries that intersect with a <i>GeoJSON</i> geometry.
<code>\$near</code>	Returns geospatial objects in proximity to a point.
<code>\$nearSphere</code>	Returns geospatial objects in proximity to a point on a sphere.

7.4.4 Indexing Query Modifiers

Name	Description
<code>\$explain</code>	Forces MongoDB to report on query execution plans. See explain() .
<code>\$hint</code>	Forces MongoDB to use a specific index. See hint()
<code>\$max</code>	Specifies a minimum exclusive upper limit for the index to use in a query. See max() .
<code>\$min</code>	Specifies a minimum inclusive lower limit for the index to use in a query. See min() .
<code>\$returnKey</code>	Forces the cursor to only return fields included in the index.
<code>\$snapshot</code>	Forces the query to use the index on the <code>_id</code> field. See snapshot() .

Replication

A *replica set* in MongoDB is a group of `mongod` processes that maintain the same data set. Replica sets provide redundancy and high availability, and are the basis for all production deployments. This section introduces replication in MongoDB as well as the components and architecture of replica sets. The section also provides tutorials for common tasks related to replica sets.

[**Replication Introduction** \(page 373\)](#) An introduction to replica sets, their behavior, operation, and use.

[**Replication Concepts** \(page 377\)](#) The core documentation of replica set operations, configurations, architectures and behaviors.

[**Replica Set Members** \(page 378\)](#) Introduces the components of replica sets.

[**Replica Set Deployment Architectures** \(page 386\)](#) Introduces architectural considerations related to replica sets deployment planning.

[**Replica Set High Availability** \(page 392\)](#) Presents the details of the automatic failover and recovery process with replica sets.

[**Replica Set Read and Write Semantics** \(page 398\)](#) Presents the semantics for targeting read and write operations to the replica set, with an awareness of location and set configuration.

[**Replica Set Tutorials** \(page 415\)](#) Tutorials for common tasks related to the use and maintenance of replica sets.

[**Replication Reference** \(page 462\)](#) Reference for functions and operations related to replica sets.

8.1 Replication Introduction

Replication is the process of synchronizing data across multiple servers.

8.1.1 Purpose of Replication

Replication provides redundancy and increases data availability. With multiple copies of data on different database servers, replication protects a database from the loss of a single server. Replication also allows you to recover from hardware failure and service interruptions. With additional copies of the data, you can dedicate one to disaster recovery, reporting, or backup.

In some cases, you can use replication to increase read capacity. Clients have the ability to send read and write operations to different servers. You can also maintain copies in different data centers to increase the locality and availability of data for distributed applications.

8.1.2 Replication in MongoDB

A replica set is a group of mongod instances that host the same data set. One mongod, the primary, receives all write operations. All other instances, secondaries, apply operations from the primary so that they have the same data set.

The **primary** accepts all write operations from clients. Replica set can have only one primary. Because only one member can accept write operations, replica sets provide **strict consistency**. To support replication, the primary logs all changes to its data sets in its *oplog* (page 406). See *primary* (page 378) for more information.

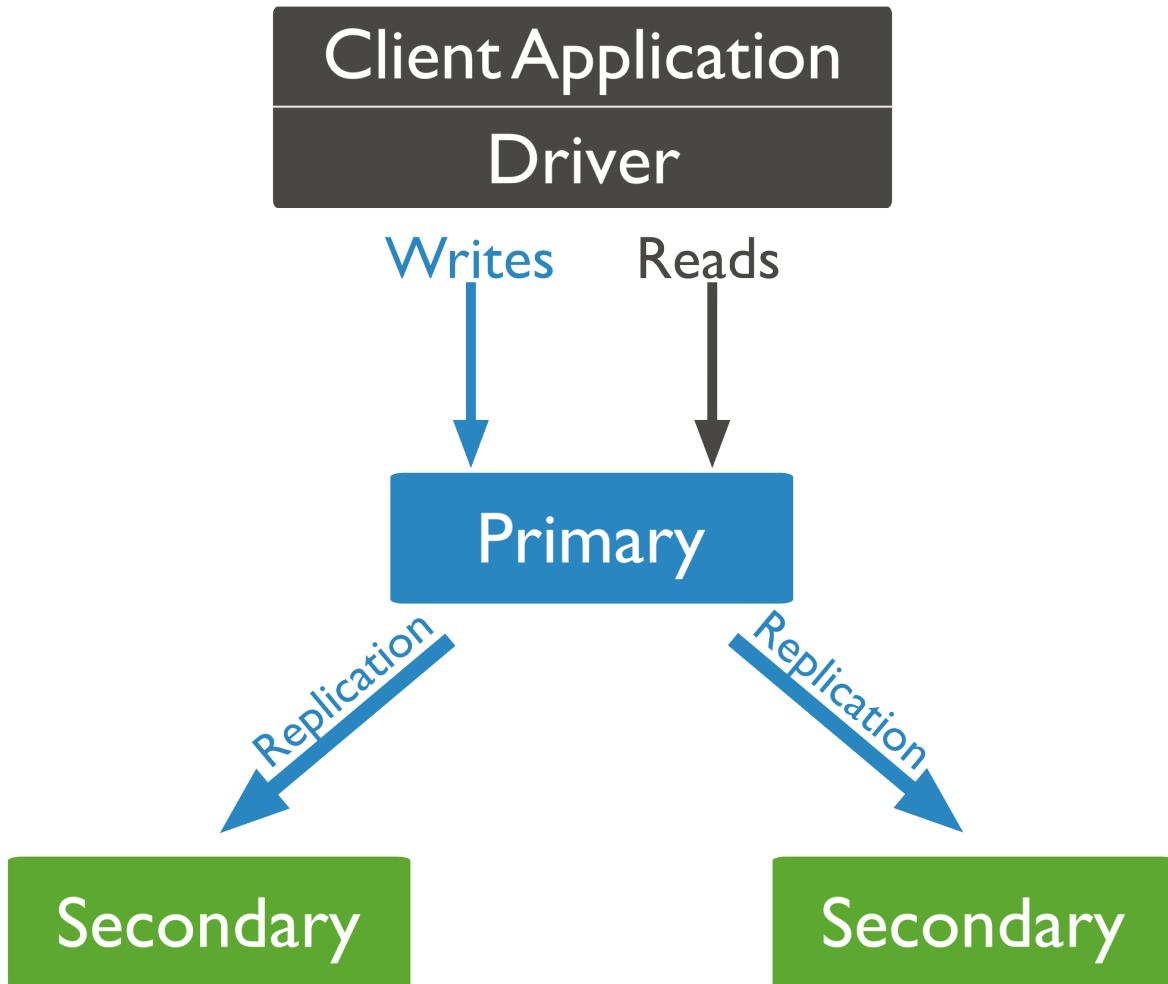


Figure 8.1: Diagram of default routing of reads and writes to the primary.

The **secondaries** replicate the primary's oplog and apply the operations to their data sets. Secondaries' data sets reflect the primary's data set. If the primary is unavailable, the replica set will elect a secondary to be primary. By default, clients read from the primary, however, clients can specify a *read preferences* (page 401) to send read operations to secondaries. See *secondaries* (page 378) for more information.

You may add an extra mongod instance a replica set as an **arbiter**. Arbiters do not maintain a data set. Arbiters only exist to vote in elections. If your replica set has an even number of members, add an arbiter to obtain a majority of votes in an election for primary. Arbiters do not require dedicated hardware. See *arbiter* (page 385) for more information.

Note: An **arbiter** will always be an arbiter. A **primary** may step down and become a **secondary**. A **secondary** may

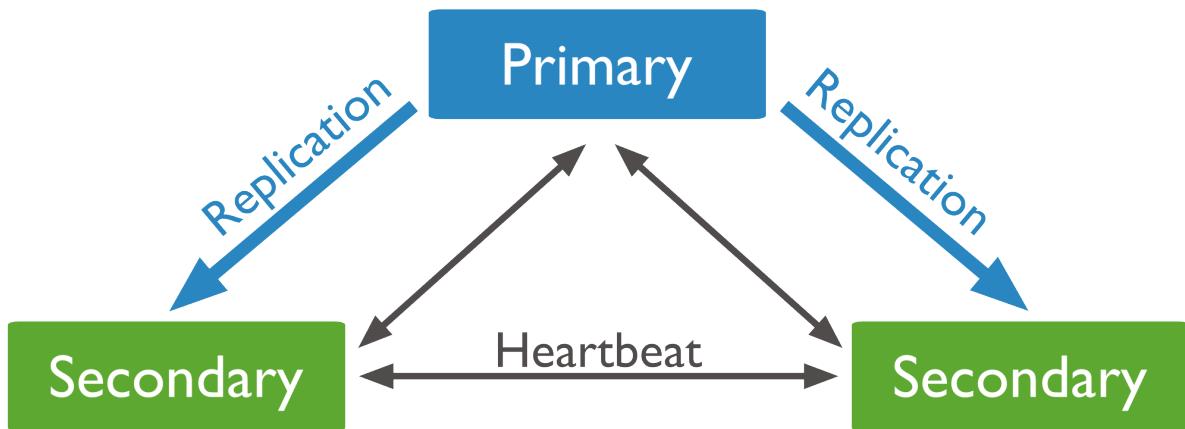


Figure 8.2: Diagram of a 3 member replica set that consists of a primary and two secondaries.

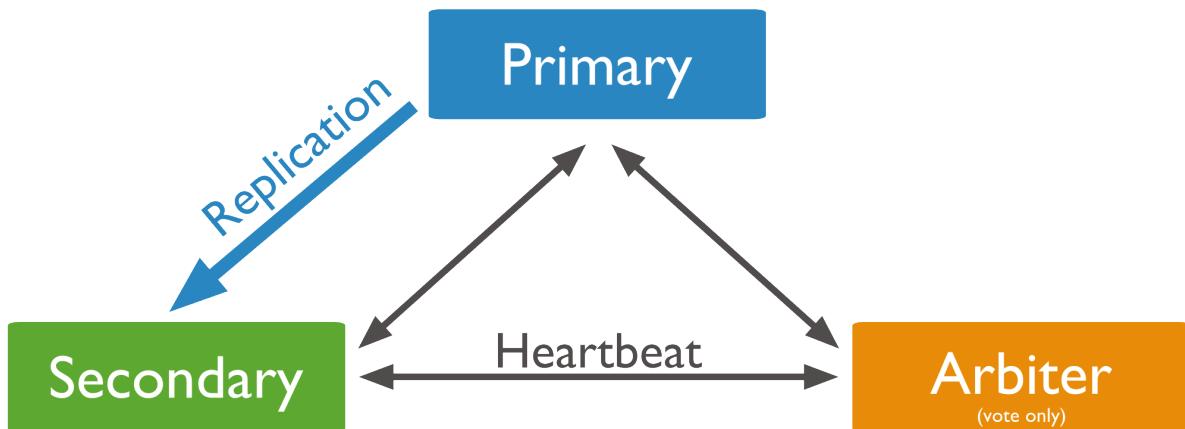


Figure 8.3: Diagram of a replica set that consists of a primary, a secondary, and an arbiter.

become the primary during an election.

Asynchronous Replication

Secondaries apply operations from the primary asynchronously. By applying operations after the primary, sets can continue to function without some members. However, as a result secondaries may not return the most current data to clients.

See [Replica Set Oplog](#) (page 406) and [Replica Set Data Synchronization](#) (page 407) for more information. See [Read Preference](#) (page 401) for more on read operations and secondaries.

Automatic Failover

When a primary does not communicate with the other members of the set for more than 10 seconds, the replica set will attempt to select another member to become the new primary. The first secondary that receives a majority of the votes becomes primary.

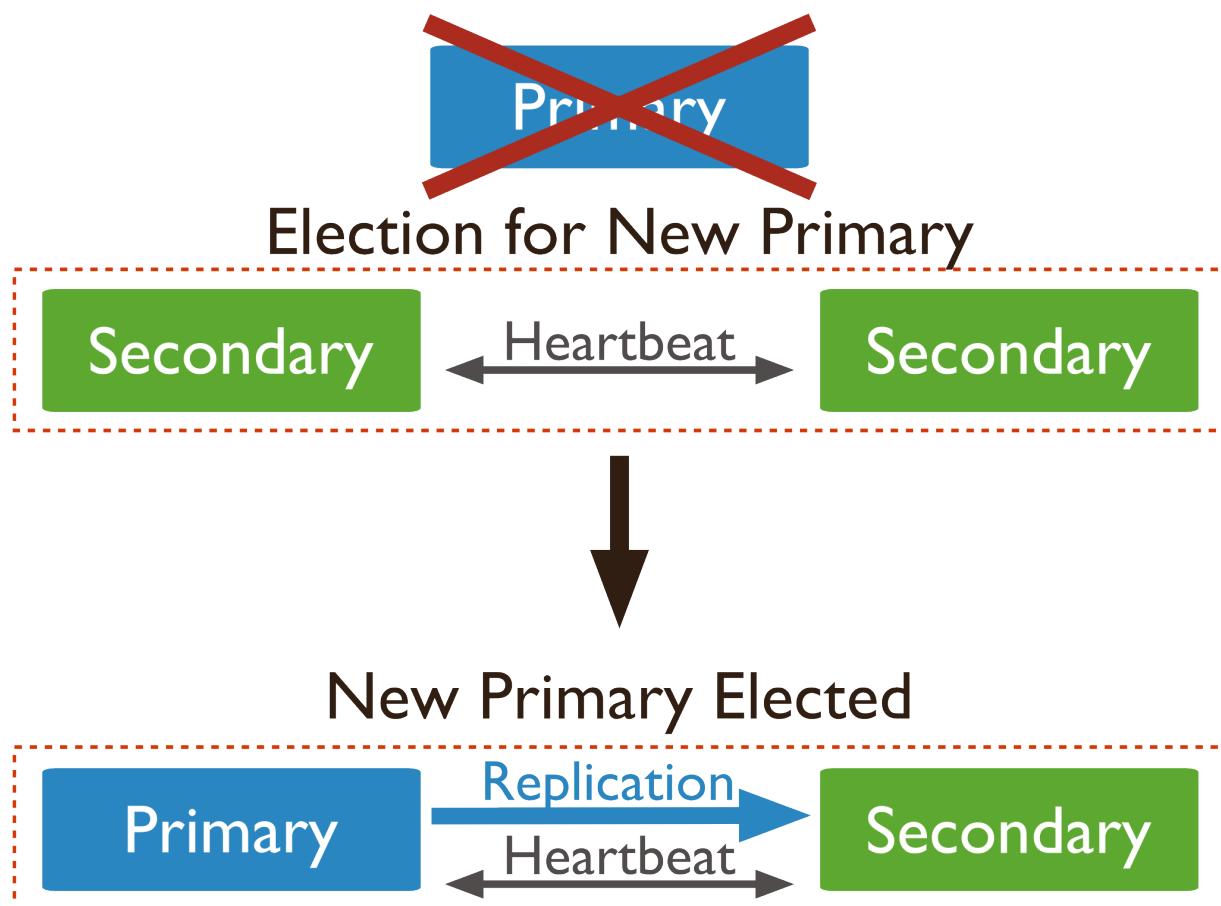


Figure 8.4: Diagram of an election of a new primary. In a three member replica set with two secondaries, the primary becomes unreachable. The loss of a primary triggers an election where one of the secondaries becomes the new primary

See [Replica Set Elections](#) (page 393) and [Rollbacks During Replica Set Failover](#) (page 397) for more information.

Additional Features

Replica sets provide a number of options to support application needs. For example, you may deploy a replica set with [members in multiple data centers](#) (page 392), or control the outcome of elections by adjusting the [priority](#) (page 476) of some members. Replica sets also support dedicated members for reporting, disaster recovery, or backup functions.

See [Priority 0 Replica Set Members](#) (page 382), [Hidden Replica Set Members](#) (page 383) and [Delayed Replica Set Members](#) (page 383) for more information.

8.2 Replication Concepts

These documents describe and provide examples of replica set operation, configuration, and behavior. For an overview of replication, see [Replication Introduction](#) (page 373). For documentation of the administration of replica sets, see [Replica Set Tutorials](#) (page 415). The [Replication Reference](#) (page 462) documents commands and operations specific to replica sets.

[Replica Set Members](#) (page 378) Introduces the components of replica sets.

[Replica Set Primary](#) (page 378) The primary is the only member of a replica set that accepts write operations.

[Replica Set Secondary Members](#) (page 378) Secondary members replicate the primary's data set and accept read operations. If the set has no primary, a secondary can become primary.

[Priority 0 Replica Set Members](#) (page 382) Priority 0 members are secondaries that cannot become the primary.

[Hidden Replica Set Members](#) (page 383) Hidden members are secondaries that are invisible to applications. These members support dedicated workloads, such as reporting or backup.

[Replica Set Arbiter](#) (page 385) An arbiter does not maintain a copy of the data set but participate in elections.

[Replica Set Deployment Architectures](#) (page 386) Introduces architectural considerations related to replica sets deployment planning.

[Three Member Replica Sets](#) (page 387) Three-member replica sets provide the minimum recommended architecture for a replica set.

[Replica Sets with Four or More Members](#) (page 388) Four or more member replica sets provide greater redundancy and can support greater distribution of read operations and dedicated functionality.

[Replica Set High Availability](#) (page 392) Presents the details of the automatic failover and recovery process with replica sets.

[Replica Set Elections](#) (page 393) Elections occur when the primary becomes unavailable and the replica set members autonomously select a new primary.

[Read Preference](#) (page 401) Applications specify *read preference* to control how drivers direct read operations to members of the replica set.

[Replication Processes](#) (page 406) Mechanics of the replication process and related topics.

[Master Slave Replication](#) (page 409) Master-slave replication provided redundancy in early versions of MongoDB. Replica sets replace master-slave for most use cases.

8.2.1 Replica Set Members

A *replica set* in MongoDB is a group of `mongod` processes that provide redundancy and high availability. The members of a replica set are:

Primary ([page ??](#)). The *primary* receives all write operations.

Secondaries ([page ??](#)). Secondaries replicate operations from the primary to maintain an identical data set. Secondaries may have additional configurations for special usage profiles. For example, secondaries may be *non-voting* ([page 396](#)) or *priority 0* ([page 382](#)).

You can also maintain an *arbiter* ([page ??](#)) as part of a replica set. Arbiters do not keep a copy of the data. However, arbiters play a role in the elections that select a primary if the current primary is unavailable.

A replica set can have up to 12 members.¹ However, only 7 members can vote at a time.

The minimum requirements for a replica set are: A *primary* ([page ??](#)), a *secondary* ([page ??](#)), and an *arbiter* ([page ??](#)). Most deployments, however, will keep three members that store data: A *primary* ([page ??](#)) and two *secondary members* ([page ??](#)).

Replica Set Primary

The primary is the only member in the replica set that receives write operations. MongoDB applies write operations on the *primary* and then records the operations on the primary's *oplog* ([page 406](#)). *Secondary* ([page ??](#)) members replicate this log and apply the operations to their data sets.

In the following three-member replica set, the primary accepts all write operations. Then the secondaries replicate the oplog to apply to their data sets.

All members of the replica set can accept read operations. However, by default, an application directs its read operations to the primary member. See [Read Preference](#) ([page 401](#)) for details on changing the default read behavior.

The replica set can have at most one primary. If the current primary becomes unavailable, an election determines the new primary. See [Replica Set Elections](#) ([page 393](#)) for more details.

In the following 3-member replica set, the primary becomes unavailable. This triggers an election which selects one of the remaining secondaries as the new primary.

Replica Set Secondary Members

A secondary maintains a copy of the *primary*'s data set. To replicate data, a secondary applies operations from the primary's *oplog* ([page 406](#)) to its own data set in an asynchronous process. A replica set can have one or more secondaries.

The following three-member replica set has two secondary members. The secondaries replicate the primary's oplog and apply the operations to their data sets.

Although clients cannot write data to secondaries, clients can read data from secondary members. See [Read Preference](#) ([page 401](#)) for more information on how clients direct read operations to replica sets.

A secondary can become a primary. If the current primary becomes unavailable, the replica set holds an *election* to choose which of the secondaries becomes the new primary.

In the following three-member replica set, the primary becomes unavailable. This triggers an election where one of the remaining secondaries becomes the new primary.

See [Replica Set Elections](#) ([page 393](#)) for more details.

¹ While replica sets are the recommended solution for production, a replica set can support only 12 members in total. If your deployment requires more than 12 members, you'll need to use [master-slave](#) ([page 409](#)) replication. Master-slave replication lacks the automatic failover capabilities.

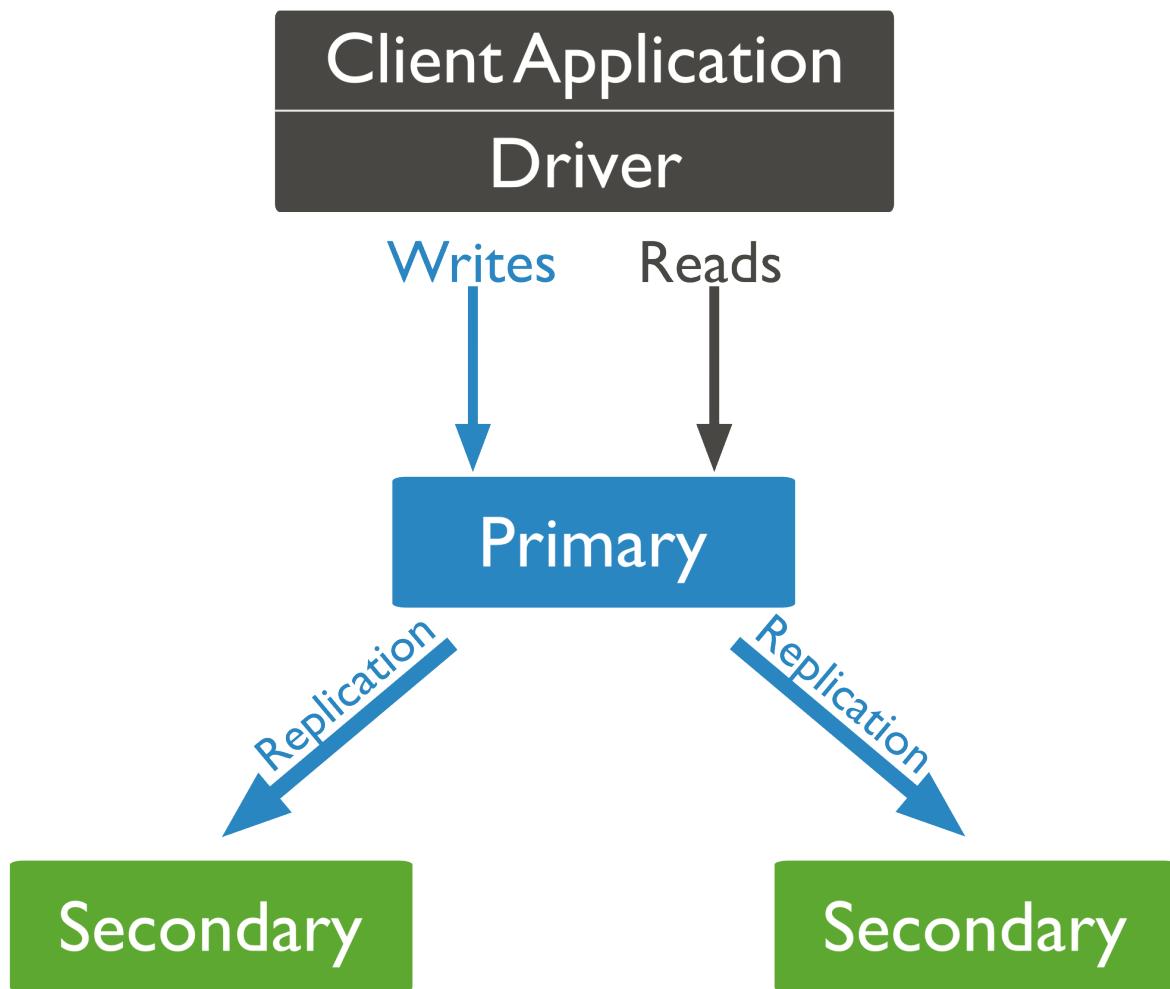


Figure 8.5: Diagram of default routing of reads and writes to the primary.

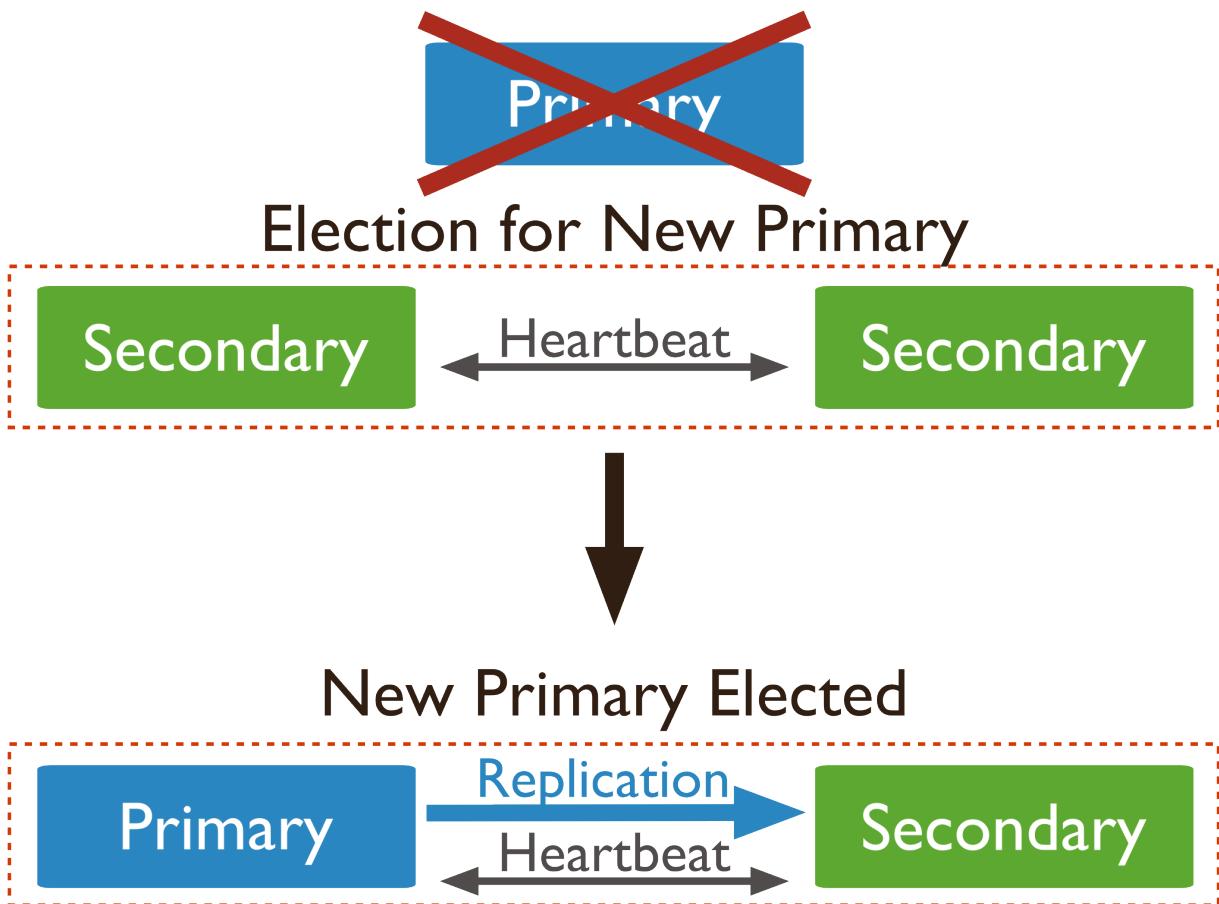


Figure 8.6: Diagram of an election of a new primary. In a three member replica set with two secondaries, the primary becomes unreachable. The loss of a primary triggers an election where one of the secondaries becomes the new primary

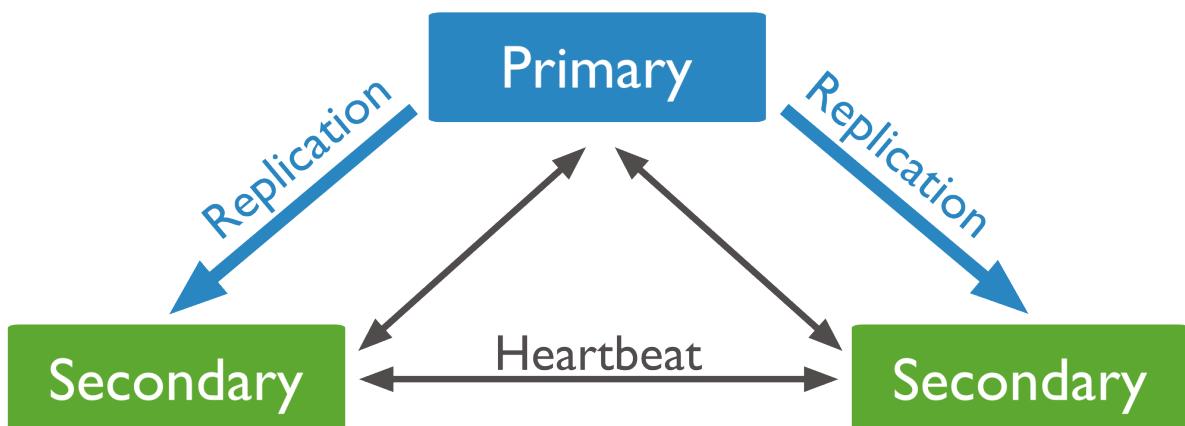


Figure 8.7: Diagram of a 3 member replica set that consists of a primary and two secondaries.

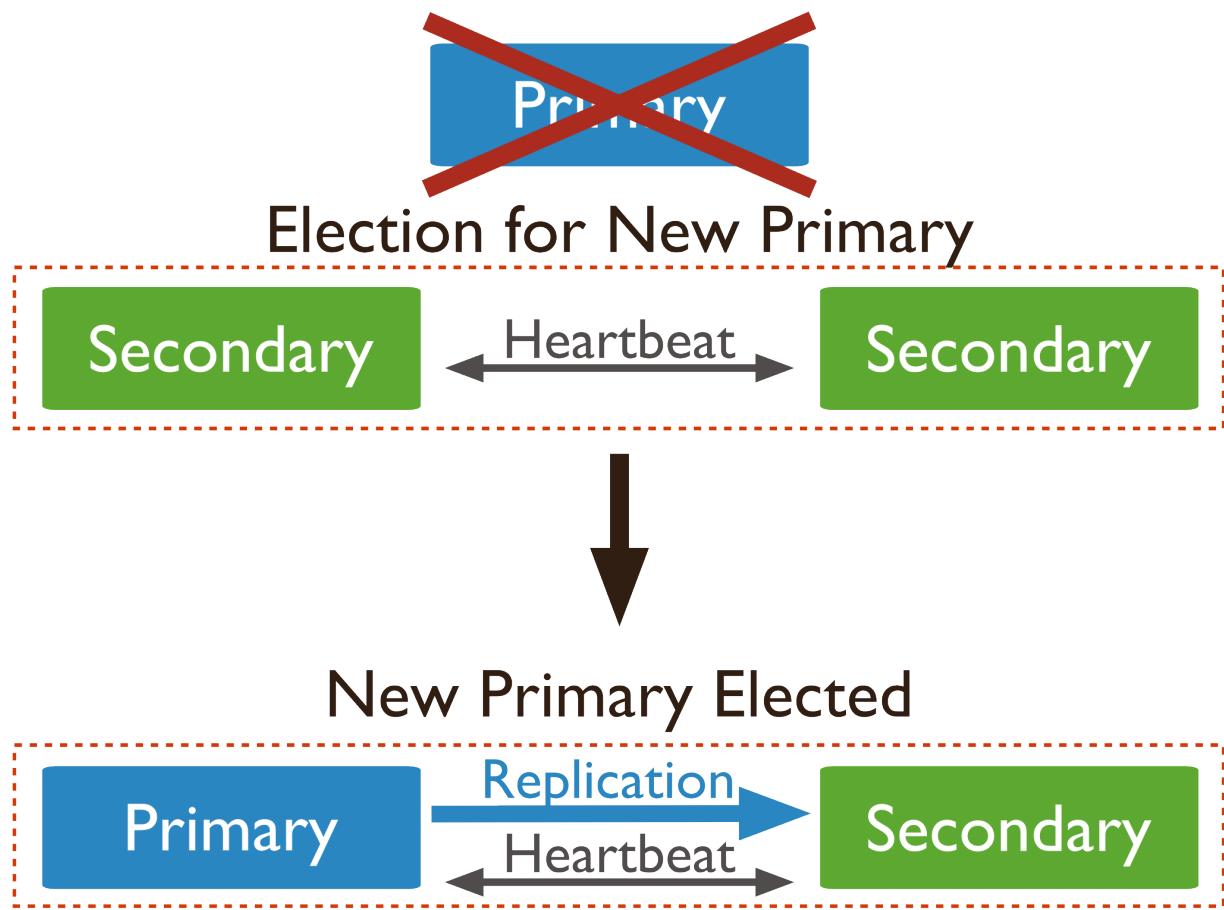


Figure 8.8: Diagram of an election of a new primary. In a three member replica set with two secondaries, the primary becomes unreachable. The loss of a primary triggers an election where one of the secondaries becomes the new primary

You can configure a secondary member for a specific purpose. You can configure a secondary to:

- Prevent it from becoming a primary in an election, which allows it to reside in a secondary data center or to serve as a cold standby. See [Priority 0 Replica Set Members](#) (page 382).
- Prevent applications from reading from it, which allows it to run applications that require separation from normal traffic. See [Hidden Replica Set Members](#) (page 383).
- Keep a running “historical” snapshot for use in recovery from certain errors, such as unintentionally deleted databases. See [Delayed Replica Set Members](#) (page 383).

Priority 0 Replica Set Members

A *priority 0* member is a secondary that **cannot** become *primary*. *Priority 0* members cannot *trigger elections*. Otherwise these members function as normal secondaries. A *priority 0* member maintains a copy of the data set, accepts read operations, and votes in elections. Configure a *priority 0* member to prevent *secondaries* from becoming primary, which is particularly useful in multi-data center deployments.

In a three-member replica set, in one data center hosts the primary and a secondary. A second data center hosts one *priority 0* member that cannot become primary.

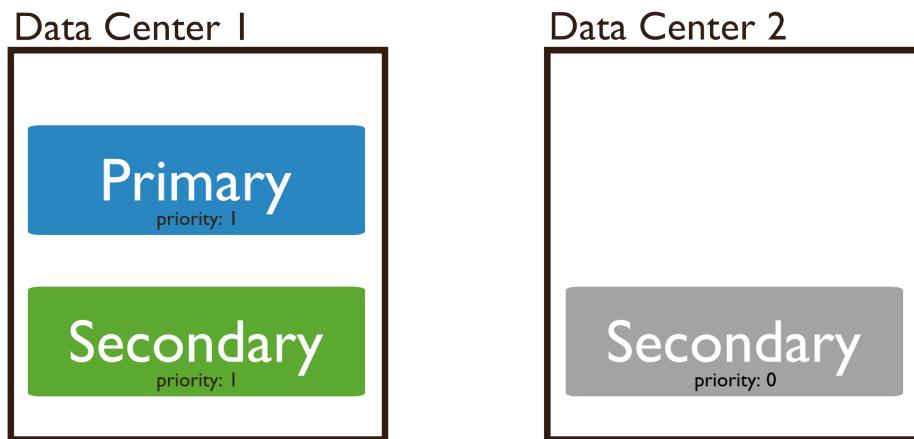


Figure 8.9: Diagram of a 3 member replica set distributed across two data centers. Replica set includes a priority 0 member.

Priority 0 Members as Standbys A *priority 0* member can function as a standby. In some replica sets, it might not be possible to add a new member in a reasonable amount of time. A standby member keeps a current copy of the data to be able to replace an unavailable member.

In many cases, you need not set standby to *priority 0*. However, in sets with varied hardware or *geographic distribution* (page 392), a *priority 0* standby ensures that only qualified members become primary.

A *priority 0* standby may also be valuable for some members of a set with different hardware or workload profiles. In these cases, deploy a member with *priority 0* so it can't become primary. Also consider using an *hidden member* (page 383) for this purpose.

If your set already has seven voting members, also configure the member as *non-voting* (page 396).

Priority 0 Members and Failover When configuring a *priority 0* member, consider potential failover patterns, including all possible network partitions. Always ensure that your main data center contains both a quorum of voting members and contains members that are eligible to be primary.

Configuration To configure a *priority 0* member, see [Prevent Secondary from Becoming Primary](#) (page 434).

Hidden Replica Set Members

A hidden member maintains a copy of the *primary*'s data set but is **invisible** to client applications. Hidden members are ideal for workloads with different usage patterns from the other members in the *replica set*. Hidden members are also [priority 0 members](#) (page 382) and **cannot become primary**. The `db.isMaster()` method does not display hidden members. Hidden members, however, **do vote** in [elections](#) (page 393).

In the following five-member replica set, all four secondary members have copies of the primary's data set, but one of the secondary members is hidden.

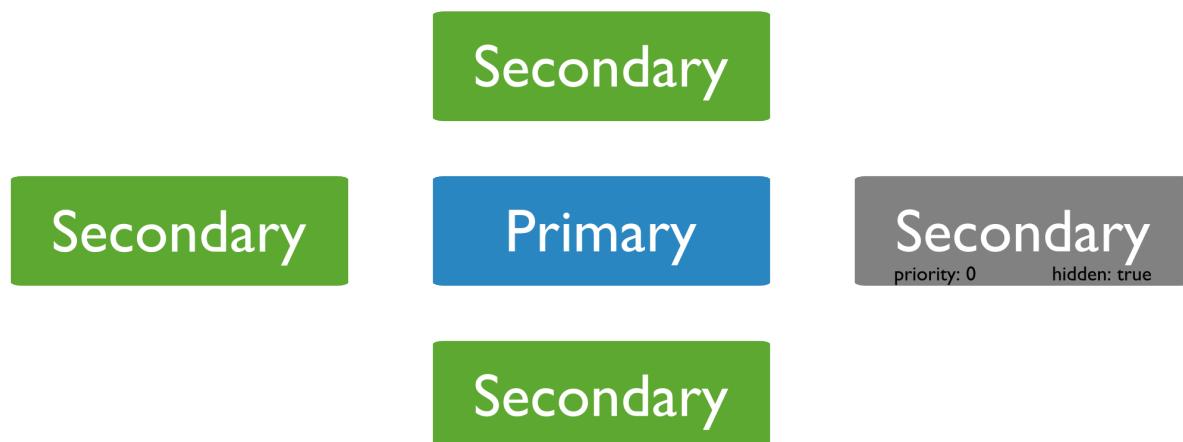


Figure 8.10: Diagram of a 5 member replica set with a hidden priority 0 member.

Secondary reads do not reach a hidden member, so the member receives no traffic beyond what replication requires. It can be useful to keep a hidden member dedicated to reporting or to do backups.

For dedicated backup, ensure that the hidden member has low network latency to the primary or likely primary. Ensure that the *replication lag* is minimal or non-existent.

Avoid stopping the `mongod` process of a hidden members. Instead, for filesystem snapshots, use `db.fsyncLock()` to flush all writes and lock the `mongod` instance for the duration of the backup.

For more information about backing up MongoDB databases, see [Backup Strategies for MongoDB Systems](#) (page 134). To configure a hidden member, see [Configure a Hidden Replica Set Member](#) (page 436).

Delayed Replica Set Members

Delayed members contain copies of a *replica set*'s data set. However, a delayed member's data set reflects an earlier, or delayed, state of the set. For example, if the current time is 09:52 and a member has a delay of an hour, the delayed member has no operation more recent than 08:52.

Because delayed members are a “rolling backup” or a running “historical” snapshot of the data set, they may help you recover from various kinds of human error. For example, a delayed member can make it possible to recover from unsuccessful application upgrades and operator errors including dropped databases and collections.

Requirements

Delayed members:

- **Must be** [priority 0](#) (page 382) members. Set the priority to 0 to prevent a delayed member from becoming primary.
- **Should be** [hidden](#) (page 383) members. Always prevent applications from seeing and querying delayed members.
- *do vote in elections* for primary.

Delayed members apply operations from the *oplog* on a delay. When choosing the amount of delay, consider that the amount of delay:

- must be equal to or greater than your maintenance windows.
- must be *smaller* than the capacity of the oplog. For more information on oplog size, see [Oplog Size](#) (page 406).

Example In the following 5-member replica set, the primary and all secondaries have copies of the data set. One member applies operations with a delay of 3600 seconds, or an hour. This delayed member is also *hidden* and is a *priority 0 member*.

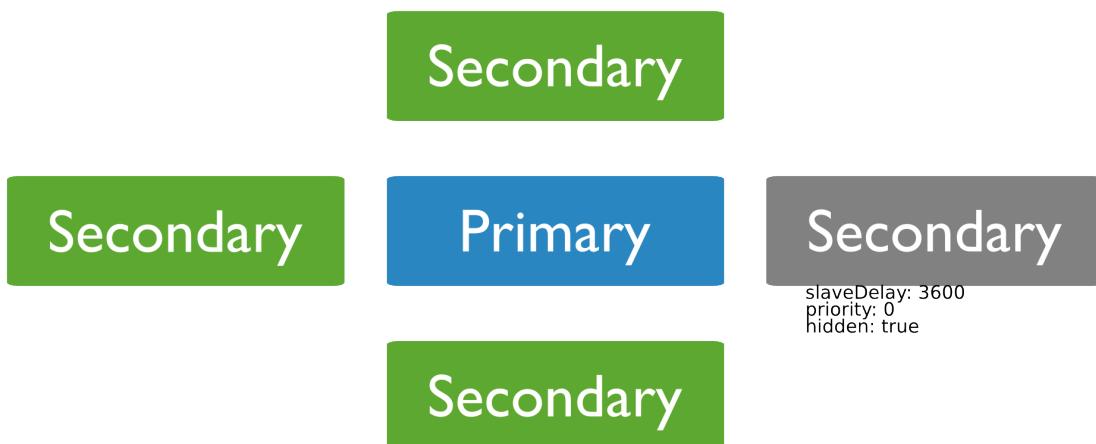


Figure 8.11: Diagram of a 5 member replica set with a hidden delayed priority 0 member.

Configuration A delayed member has its [priority](#) (page 476) equal to 0, [hidden](#) (page 475) equal to true, and its [slaveDelay](#) (page 476) equal to the number of seconds of delay:

```
{
  "_id" : <num>,
  "host" : <hostname:port>,
  "priority" : 0,
  "slaveDelay" : <seconds>,
  "hidden" : true
}
```

To configure a delayed member, see [Configure a Delayed Replica Set Member](#) (page 437).

Replica Set Arbiter

An arbiter does **not** have a copy of data set and **cannot** become a primary. Replica sets may have arbiters to add a vote in [elections of or for primary](#) (page 393). Arbiters allow replica sets to have an uneven number of members, without the overhead of a member that replicates data.

Important: Do not run an arbiter on systems that also host the primary or the secondary members of the replica set.

Only add an arbiter to sets with even numbers of members. If you add an arbiter to a set with an odd number of members, the set may suffer from tied *elections*. To add an arbiter, see [Add an Arbiter to Replica Set](#) (page 427).

Example

For example, in the following replica set, an arbiter allows the set to have an odd number of votes for elections:



Figure 8.12: Diagram of a four member replica set plus an arbiter for odd number of votes.

Security

Authentication When running with auth, arbiters exchange credentials with other members of the set to authenticate. MongoDB encrypts the authentication process. The MongoDB authentication exchange is cryptographically secure.

Arbiters, use keyfiles to authenticate to the replica set.

Communication The only communication between arbiters and other set members are: votes during elections, heartbeats, and configuration data. These exchanges are not encrypted.

However, if your MongoDB deployment uses SSL, MongoDB will encrypt *all* communication between replica set members. See [Connect to MongoDB with SSL](#) (page 248) for more information.

As with all MongoDB components, run arbiters on in trusted network environments.

8.2.2 Replica Set Deployment Architectures

The architecture of a *replica set* affects the set's capacity and capability. This document provides strategies for replica set deployments and describes common architectures.

The standard replica set deployment for production system is a three-member replica set. These sets provide redundancy and fault tolerance. Avoid complexity when possible, but let your application requirements dictate the architecture.

Important: 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.

Strategies

Determine the Number of Members

Add members in a replica set according to these strategies.

Deploy an Odd Number of Members An odd number of members ensures that the replica set is always able to elect a primary. If you have an even number of members, add an arbiter to get an odd number. *Arbiters* do not store a copy of the data and require fewer resources. As a result, you may run an arbiter on an application server or other shared process.

Consider Fault Tolerance *Fault tolerance* for a replica set is the number of members that can become unavailable and still leave enough members in the set to elect a primary. In other words, it is the difference between the number of members in the set and the majority needed to elect a primary. Without a primary, a replica set cannot accept write operations. Fault tolerance is an effect of replica set size, but the relationship is not direct. See the following table:

Number of Members.	Majority Required to Elect a New Primary.	Fault Tolerance.
3	2	1
4	3	1
5	3	2
6	4	2

Adding a member to the replica set does not *always* increase the fault tolerance. However, in these cases, additional members can provide support for dedicated functions, such as backups or reporting.

Use Hidden and Delayed Members for Dedicated Functions Add [hidden](#) (page 383) or [delayed](#) (page 383) members to support dedicated functions, such as backup or reporting.

Load Balance on Read-Heavy Deployments In a deployment with *very* high read traffic, you can improve read throughput by distributing reads to secondary members. As your deployment grows, add or move members to alternate data centers to improve redundancy and availability.

Always ensure that the main facility is able to elect a primary.

Add Capacity Ahead of Demand The existing members of a replica set must have spare capacity to support adding a new member. Always add new members before the current demand saturates the capacity of the set.

Determine the Distribution of Members

Distribute Members Geographically To protect your data if your main data center fails, keep at least one member in an alternate data center. Set these members' [priority](#) (page 476) to 0 to prevent them from becoming primary.

Keep a Majority of Members in One Location When a replica set has members in multiple data centers, network partitions can prevent communication between data centers. To replicate data, members must be able to communicate to other members.

In an election, members must see each other to create a majority. To ensure that the replica set members can confirm a majority and elect a primary, keep a majority of the set's members in one location.

Target Operations with Tags

Use [replica set tags](#) (page 446) to ensure that operations replicate to specific data centers. Tags also support targeting read operations to specific machines.

See also:

[Data Center Awareness](#) (page 157) and [Operational Segregation in MongoDB Deployments](#) (page 157).

Use Journaling to Protect Against Power Failures

Enable journaling to protect data against service interruptions. Without journaling MongoDB cannot recover data after unexpected shutdowns, including power failures and unexpected reboots.

All 64-bit versions of MongoDB after version 2.0 have journaling enabled by default.

Deployment Patterns

The following documents describe common replica set deployment patterns. Other patterns are possible and effective depending on the application's requirements. If needed, combine features of each architecture in your own deployment:

[Three Member Replica Sets](#) (page 387) Three-member replica sets provide the minimum recommended architecture for a replica set.

[Replica Sets with Four or More Members](#) (page 388) Four or more member replica sets provide greater redundancy and can support greater distribution of read operations and dedicated functionality.

[Geographically Distributed Replica Sets](#) (page 392) Geographically distributed sets include members in multiple locations to protect against facility-specific failures, such as power outages.

Three Member Replica Sets

The minimum architecture of a replica set has three members. A three member replica set can have either three members that hold data, or two members that hold data and an arbiter.

Primary with Two Secondary Members A replica set with three members that store data has:

- One [primary](#) (page 378).
- Two [secondary](#) (page 378) members. Both secondaries can become the primary in an [election](#) (page 393).

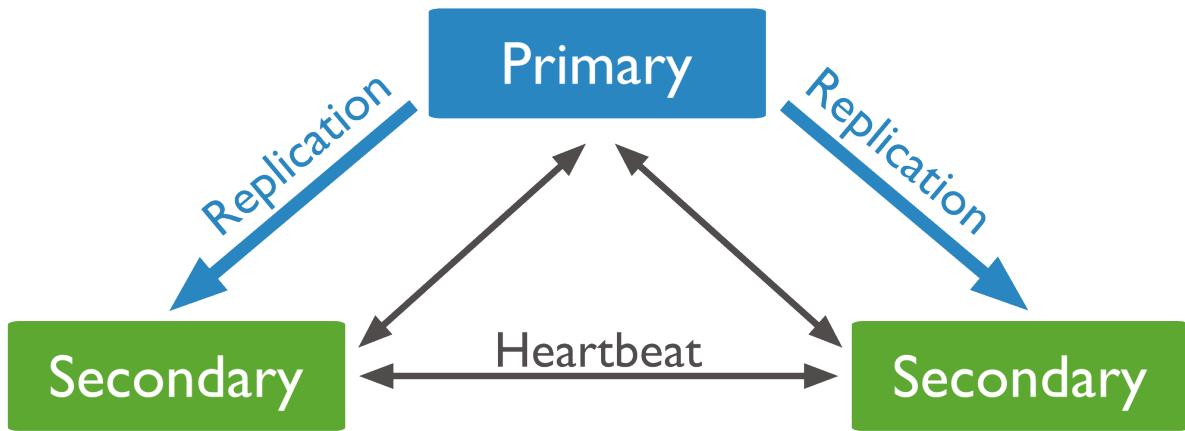


Figure 8.13: Diagram of a 3 member replica set that consists of a primary and two secondaries.

These deployments provide two complete copies of the data set at all times in addition to the primary. These replica sets provide additional fault tolerance and [high availability](#) (page 392). If the primary is unavailable, the replica set elects a secondary to be primary and continues normal operation. The old primary rejoins the set when available.

Primary with a Secondary and an Arbiter A three member replica set with a two members that store data has:

- One [primary](#) (page 378).
- One [secondary](#) (page 378) member. The secondary can become primary in an [election](#) (page 393).
- One [arbiter](#) (page 385). The arbiter only votes in elections.

Since the arbiter does not hold a copy of the data, these deployments provides only one complete copy of the data. Arbiters require fewer resources, at the expense of more limited redundancy and fault tolerance.

However, a deployment with a primary, secondary, and an arbiter ensures that a replica set remains available if the primary *or* the secondary is unavailable. If the primary is unavailable, the replica set will elect the secondary to be primary.

See also:

[Deploy a Replica Set](#) (page 416).

Replica Sets with Four or More Members

Although the standard replica set configuration has three members you can deploy larger sets. Add additional members to a set to increase redundancy or to add capacity for distributing secondary read operations.

When adding members, ensure that:

- The set has an odd number of voting members. If you have an *even* number of voting members, deploy an [arbiter](#) (page ??) so that the set has an odd number.

The following replica set needs an arbiter to have an odd number of voting members.

- A replica set can have up to 12 members,² but only 7 voting members. See [non-voting members](#) (page 396) for more information.

² While replica sets are the recommended solution for production, a replica set can support only 12 members in total. If your deployment requires more than 12 members, you'll need to use [master-slave](#) (page 409) replication. Master-slave replication lacks the automatic failover capabilities.

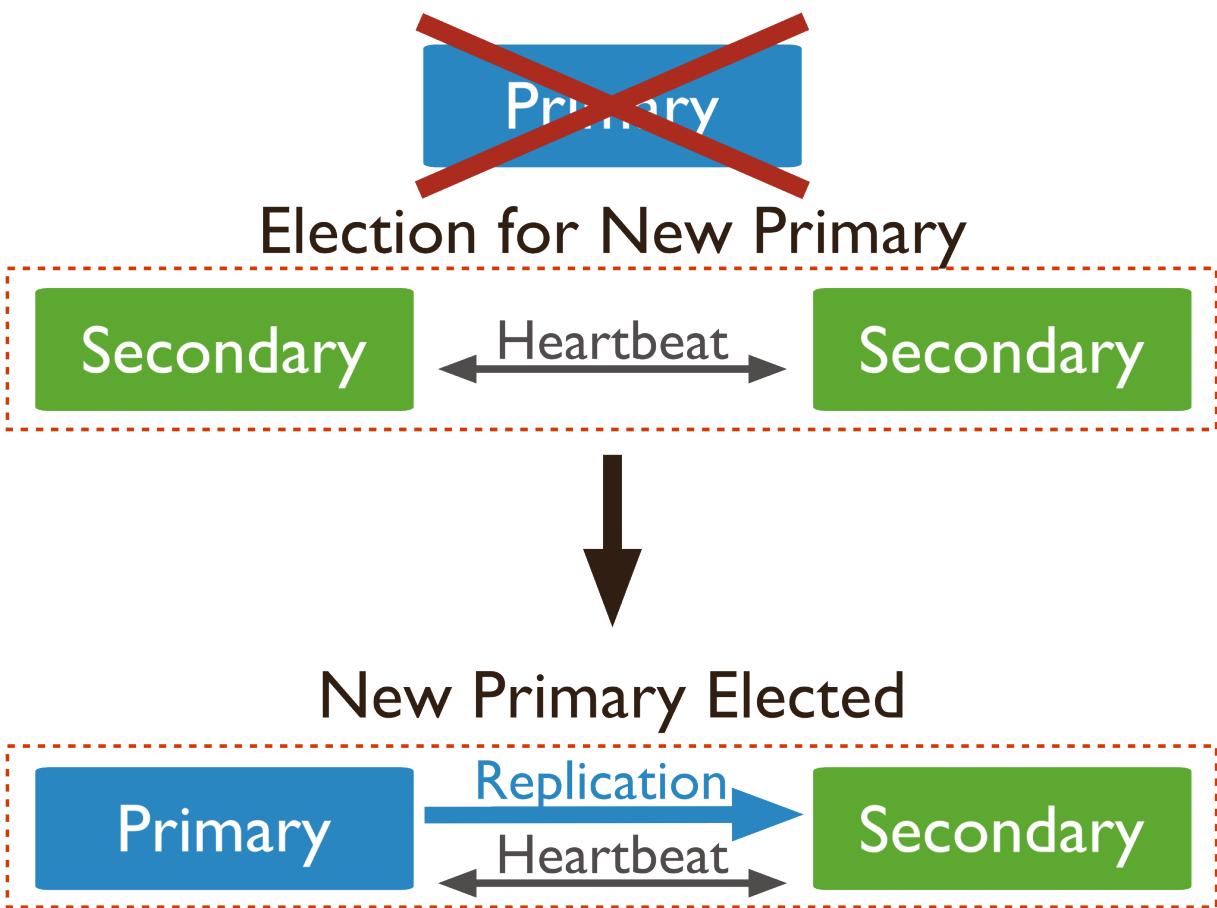


Figure 8.14: Diagram of an election of a new primary. In a three member replica set with two secondaries, the primary becomes unreachable. The loss of a primary triggers an election where one of the secondaries becomes the new primary

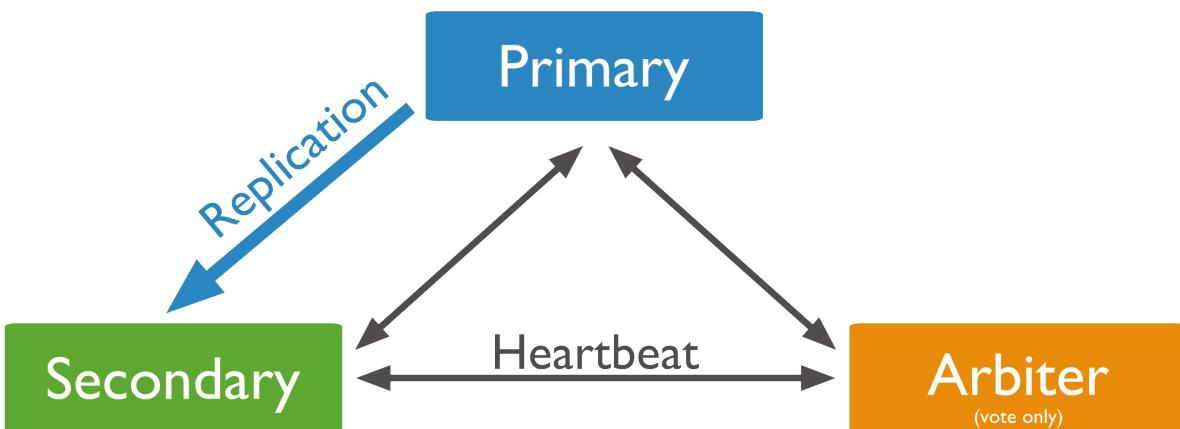


Figure 8.15: Diagram of a replica set that consists of a primary, a secondary, and an arbiter.

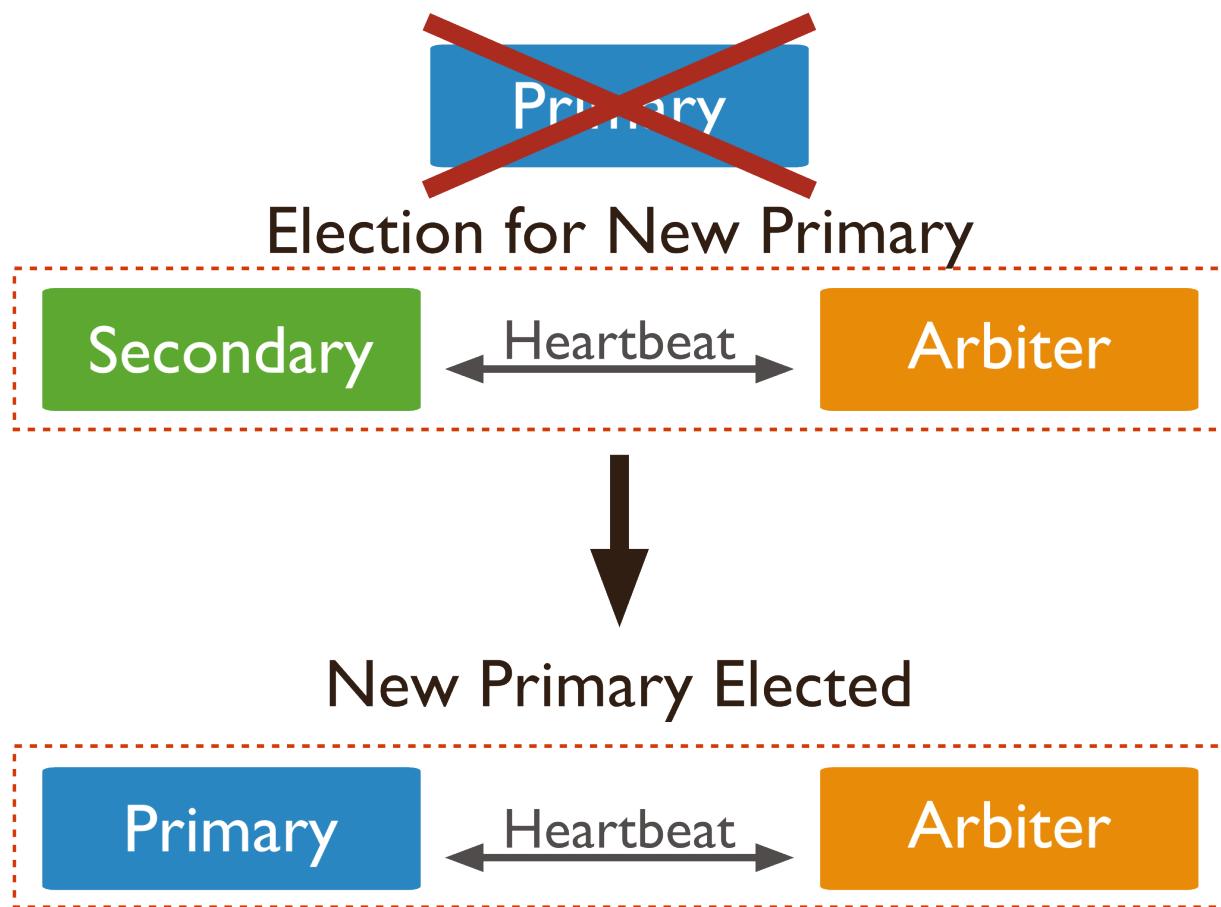


Figure 8.16: Diagram of an election of a new primary. In a three member replica set with a secondary and an arbiter, the primary becomes unreachable. The loss of a primary triggers an election where the secondary becomes new primary.



Figure 8.17: Diagram of a four member replica set plus an arbiter for odd number of votes.

The following 9 member replica set has 7 voting members and 2 non-voting members.

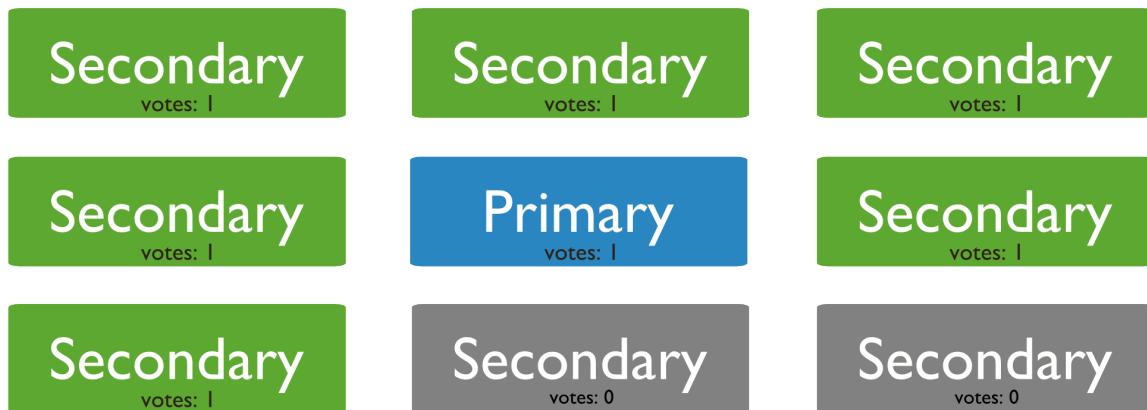


Figure 8.18: Diagram of a 9 member replica set with the maximum of 7 voting members.

- Members that cannot become primary in a *failover* have [priority 0 configuration](#) (page 382).

For instance, some members that have limited resources or networking constraints and should never be able to become primary. Configure members that should not become primary to have [priority 0](#) (page 382). In following replica set, the secondary member in the third data center has a priority of 0:

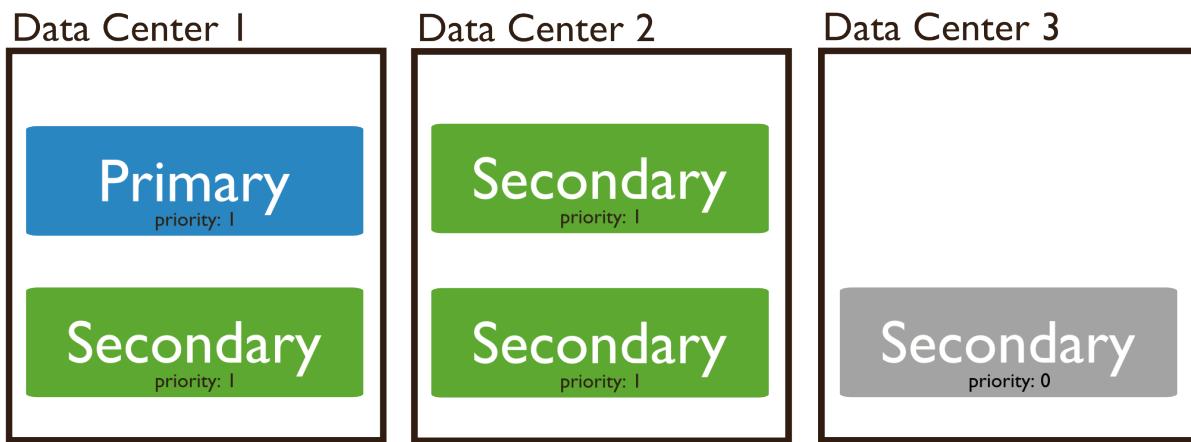


Figure 8.19: Diagram of a 5 member replica set distributed across three data centers. Replica set includes a priority 0 member.

- A majority of the set's members should be in your applications main data center.

See also:

[Deploy a Replica Set](#) (page 416), [Add an Arbiter to Replica Set](#) (page 427), and [Add Members to a Replica Set](#) (page 429).

Geographically Distributed Replica Sets

Adding members to a replica set in multiple data centers adds redundancy and provides fault tolerance if one data center is unavailable. Members in additional data centers should have a [priority of 0](#) (page 382) to prevent them from becoming primary.

For example: the architecture of a geographically distributed replica set may be:

- One *primary* in the main data center.
- One *secondary* member in the main data center. This member can become primary at any time.
- One [priority 0](#) (page 382) member in a second data center. This member cannot become primary.

In the following replica set, the primary and one secondary are in *Data Center 1*, while *Data Center 2* has a [priority 0](#) (page 382) secondary that cannot become a primary.

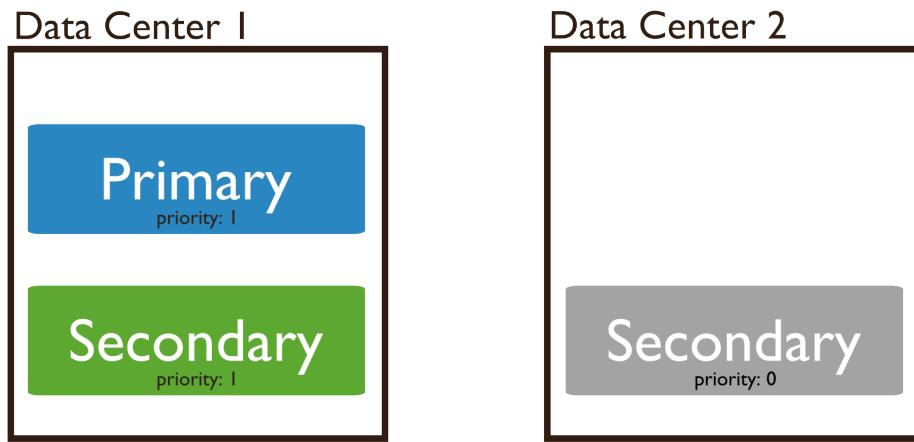


Figure 8.20: Diagram of a 3 member replica set distributed across two data centers. Replica set includes a priority 0 member.

If the primary is unavailable, the replica set will elect a new primary from *Data Center 1*. If the data centers cannot connect to each other, the member in *Data Center 2* will not become the primary.

If *Data Center 1* becomes unavailable, you can manually recover the data set from *Data Center 2* with minimal downtime. With sufficient [write concern](#) (page 44), there will be no data loss.

To facilitate elections, the main data center should hold a majority of members. Also ensure that the set has an odd number of members. If adding a member in another data center results in a set with an even number of members, deploy an [arbiter](#) (page ??). For more information on elections, see [Replica Set Elections](#) (page 393).

See also:

[Deploy a Geographically Redundant Replica Set](#) (page 421).

8.2.3 Replica Set High Availability

Replica sets provide high availability using automatic *failover*. Failover allows a *secondary* members to become *primary* if primary is unavailable. Failover, in most situations does not require manual intervention.

Replica set members keep the same data set but are otherwise independent. If the primary becomes unavailable, the replica set holds an [election](#) (page 393) to select a new primary. In some situations, the failover process may require a

[rollback](#) (page 397).³

The deployment of a replica set affects the outcome of failover situations. To support effective failover, ensure that one facility can elect a primary if needed. Choose the facility that hosts the core application systems to host the majority of the replica set. Place a majority of voting members and all the members that can become primary in this facility. Otherwise, network partitions could prevent the set from being able to form a majority.

Failover Processes

The replica set recovers from the loss of a primary by holding an election. Consider the following:

Replica Set Elections (page 393) Elections occur when the primary becomes unavailable and the replica set members autonomously select a new primary.

Rollbacks During Replica Set Failover (page 397) A rollback reverts write operations on a former primary when the member rejoins the replica set after a failover.

Replica Set Elections

Replica sets use elections to determine which set member will become *primary*. Elections occur after initiating a replica set, and also any time the primary becomes unavailable. The primary is the only member in the set that can accept write operations. If a primary becomes unavailable, elections allow the set to recover normal operations without manual intervention. Elections are part of the [failover process](#) (page 392).

Important: Elections are essential for independent operation of a replica set; however, elections take time to complete. While an election is in process, the replica set has no primary and cannot accept writes. MongoDB avoids elections unless necessary.

In the following three-member replica set, the primary is unavailable. The remaining secondaries hold an election to choose a new primary.

Factors and Conditions that Affect Elections

Heartbeats Replica set members send heartbeats (pings) to each other every two seconds. If a heartbeat does not return within 10 seconds, the other members mark the delinquent member as inaccessible.

Priority Comparisons The [priority](#) (page 476) setting affects elections. Members will prefer to vote for members with the highest priority value.

Members with a priority value of 0 cannot become primary and do not seek election. For details, see [Priority 0 Replica Set Members](#) (page 382).

A replica set does *not* hold an election as long as the current primary has the highest priority value and is within 10 seconds of the latest *oplog* entry in the set. If a higher-priority member catches up to within 10 seconds of the latest oplog entry of the current primary, the set holds an election in order to provide the higher-priority node a chance to become primary.

Optime The *optime* is the timestamp of the last operation that a member applied from the oplog. A replica set member cannot become primary unless it has the highest (i.e. most recent) *optime* of any visible member in the set.

³ Replica sets remove “rollback” data when needed without intervention. Administrators must apply or discard rollback data manually.

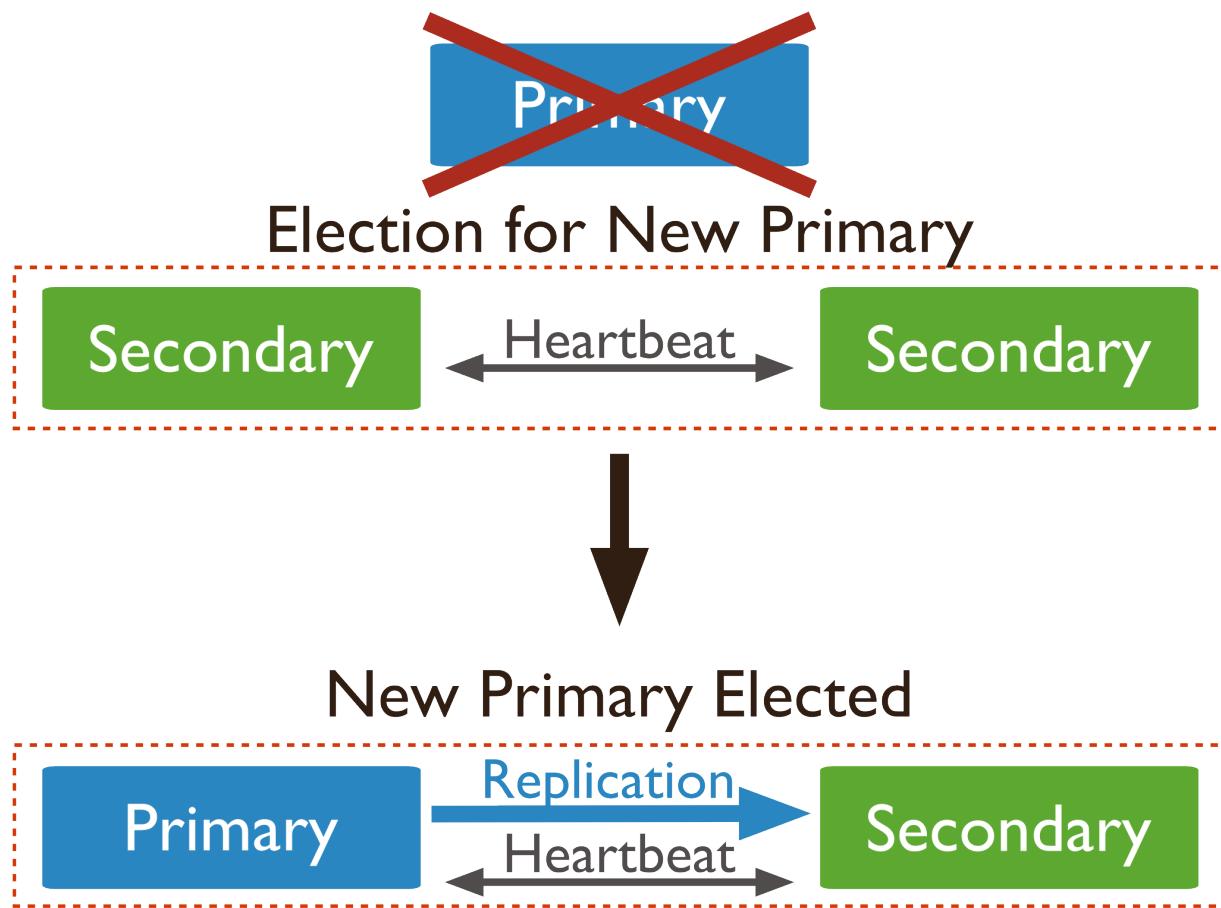


Figure 8.21: Diagram of an election of a new primary. In a three member replica set with two secondaries, the primary becomes unreachable. The loss of a primary triggers an election where one of the secondaries becomes the new primary

Connections A replica set member cannot become primary unless it can connect to a majority of the members in the replica set. For the purposes of elections, a majority refers to the total number of *votes*, rather than the total number of members.

If you have a three-member replica set, where every member has one vote, the set can elect a primary as long as two members can connect to each other. If two members are unavailable, the remaining member remains a *secondary* because it cannot connect to a majority of the set's members. If the remaining member is a *primary* and two members become unavailable, the primary steps down and becomes a secondary.

Network Partitions Network partitions affect the formation of a majority for an election. If a primary steps down and neither portion of the replica set has a majority the set will **not** elect a new primary. The replica set becomes read-only.

To avoid this situation, place a majority of instances in one data center and a minority of instances in any other data centers combined.

Election Mechanics

Election Triggering Events Replica sets hold an election any time there is no primary. Specifically, the following:

- the initiation of a new replica set.
- a secondary loses contact with a primary. Secondaries call for elections when they cannot see a primary.
- a primary steps down.

Note: *Priority 0 members* (page 382), do not trigger elections, even when they cannot connect to the primary.

A primary will step down:

- after receiving the `replSetStepDown` command.
- if one of the current secondaries is eligible for election *and* has a higher priority.
- if primary cannot contact a majority of the members of the replica set.

Important: When a primary steps down, it closes all open client connections, so that clients don't attempt to write data to a secondary. This helps clients maintain an accurate view of the replica set and helps prevent *rollbacks*.

Participation in Elections Every replica set member has a *priority* that helps determine its eligibility to become a *primary*. In an election, the replica set elects an eligible member with the highest *priority* (page 476) value as primary. By default, all members have a priority of 1 and have an equal chance of becoming primary. In the default, all members also can trigger an election.

You can set the *priority* (page 476) value to weight the election in favor of a particular member or group of members. For example, if you have a *geographically distributed replica set* (page 392), you can adjust priorities so that only members in a specific data center can become primary.

The first member to receive the majority of votes becomes primary. By default, all members have a single vote, unless you modify the *votes* (page 476) setting. *Non-voting members* (page 438) have *votes* (page 476) value of 0.

The state of a member also affects its eligibility to vote. Only members in the following states can vote: PRIMARY, SECONDARY, RECOVERING, ARBITER, and ROLLBACK.

Important: Do not alter the number of votes in a replica set to control the outcome of an election. Instead, modify the *priority* (page 476) value.

Veto in Elections All members of a replica set can veto an election, including *non-voting members* (page 396). A member will veto an election:

- If the member seeking an election is not a member of the voter's set.
- If the member seeking an election is not up-to-date with the most recent operation accessible in the replica set.
- If the member seeking an election has a lower priority than another member in the set that is also eligible for election.
- If a *priority 0 member* (page 382)⁴ is the most current member at the time of the election. In this case, another eligible member of the set will catch up to the state of this secondary member and then attempt to become primary.
- If the current primary has more recent operations (i.e. a higher `optime`) than the member seeking election, from the perspective of the voting member.
- If the current primary has the same or more recent operations (i.e. a higher or equal `optime`) than the member seeking election.

Non-Voting Members Non-voting members hold copies of the replica set's data and can accept read operations from client applications. Non-voting members do not vote in elections, but **can veto** (page 396) an election and become primary.

Because a replica set can have up to 12 members but only up to seven voting members, non-voting members allow a replica set to have more than seven members.

For instance, the following nine-member replica set has seven voting members and two non-voting members.

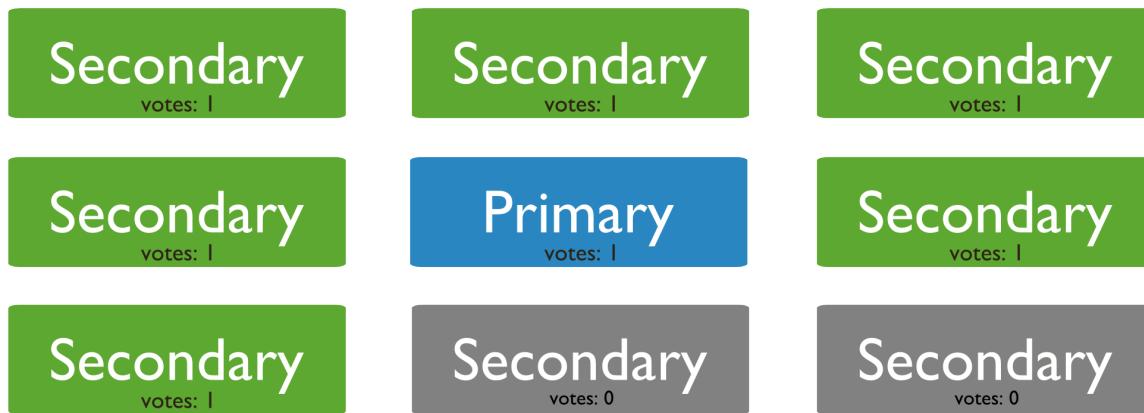


Figure 8.22: Diagram of a 9 member replica set with the maximum of 7 voting members.

A non-voting member has a `votes` (page 476) setting equal to 0 in its member configuration:

```
{  
  "_id" : <num>  
  "host" : <hostname:port>,  
  "votes" : 0  
}
```

⁴ Remember that *hidden* (page 383) and *delayed* (page 383) imply *priority 0* (page 382) configuration.

Important: Do **not** alter the number of votes to control which members will become primary. Instead, modify the [priority](#) (page 476) option. Only alter the number of votes in exceptional cases. For example, to permit more than seven members.

When possible, all members should have only one vote. Changing the number of votes can cause ties, deadlocks, and the wrong members to become primary.

To configure a non-voting member, see [Configure Non-Voting Replica Set Member](#) (page 438).

Rollbacks During Replica Set Failover

A rollback reverts write operations on a former *primary* when the member rejoins its *replica set* after a *failover*. A rollback is necessary only if the primary had accepted write operations that the *secondaries* had **not** successfully replicated before the primary stepped down. When the primary rejoins the set as a secondary, it reverts, or “rolls back,” its write operations to maintain database consistency with the other members.

MongoDB attempts to avoid rollbacks, which should be rare. When a rollback does occur, it is often the result of a network partition. Secondaries that can not keep up with the throughput of operations on the former primary, increase the size an impact of the rollback.

A rollback does *not* occur if the write operations replicate to another member of the replica set before the primary steps down and if that member remains available and accessible to a majority of the replica set.

Collect Rollback Data When a rollback does occur, administrators must decide whether to apply or ignore the rollback data. MongoDB writes the rollback data to *BSON* files in the `rollback/` folder under the database’s `dbpath` directory. The names of rollback files have the following form:

```
<database>. <collection>. <timestamp>. bson
```

For example:

```
records.accounts.2011-05-09T18-10-04.0.bson
```

Administrators must apply rollback data manually after the member completes the rollback and returns to secondary status. Use `bsondump` to read the contents of the rollback files. Then use `mongorestore` to apply the changes to the new primary.

Avoid Replica Set Rollbacks To prevent rollbacks, use [replica acknowledged write concern](#) (page 47) to guarantee that the write operations propagate to the members of a replica set.

Rollback Limitations A `mongod` instance will not rollback more than 300 megabytes of data. If your system must rollback more than 300 megabytes, you must manually intervene to recover the data. If this is the case, the following line will appear in your `mongod` log:

```
[replica set sync] replSet syncThread: 13410 replSet too much data to roll back
```

In this situation, save the data directly or force the member to perform an initial sync. To force initial sync, sync from a “current” member of the set by deleting the content of the `dbpath` directory for the member that requires a larger rollback.

See also:

[Replica Set High Availability](#) (page 392) and [Replica Set Elections](#) (page 393).

8.2.4 Replica Set Read and Write Semantics

From the perspective of a client application, whether a MongoDB instance is running as a single server (i.e. “standalone”) or a *replica set* is transparent.

By default, in MongoDB, read operations to a replica set return results from the *primary* (page 378) and are *consistent* with the last write operation.

Users may configure *read preference* on a per-connection basis to prefer that the read operations return on the *secondary* members. If clients configure the *read preference* to permit secondary reads, read operations cannot return from *secondary* members that have not replicated more recent updates or operations. When reading from a secondary, a query may return data that reflects a previous state.

This behavior is sometimes characterized as *eventual consistency* because the secondary member’s state will *eventually* reflect the primary’s state and MongoDB cannot guarantee *strict consistency* for read operations from secondary members.

To guarantee consistency for reads from secondary members, you can configure the *client* and *driver* to ensure that write operations succeed on all members before completing successfully. See [Write Concern](#) (page 44) for more information. Additionally, such configuration can help prevent [Rollbacks During Replica Set Failover](#) (page 397) during a failover.

Note: Sharded clusters where the shards are also replica sets provide the same operational semantics with regards to write and read operations.

[Write Concern for Replica Sets](#) (page 398) Write concern is the guarantee an application requires from MongoDB to consider a write operation successful.

[Read Preference](#) (page 401) Applications specify *read preference* to control how drivers direct read operations to members of the replica set.

[Read Preference Processes](#) (page 404) With replica sets, read operations may have additional semantics and behavior.

Write Concern for Replica Sets

MongoDB’s built-in [write concern](#) (page 44) confirms the success of write operations to a *replica set’s primary*. Write concern uses the `getLastError` command after write operations to return an object with error information or confirmation that there are no errors.

From the perspective of a client application, whether a MongoDB instance is running as a single server (i.e. “standalone”) or a *replica set* is transparent. However, replica sets offer some configuration options for write and read operations.⁵

Verify Write Operations

The default write concern confirms write operations only on the primary. You can configure write concern to confirm write operations to additional replica set members as well by issuing the `getLastError` command with the `w` option.

The `w` option confirms that write operations have replicated to the specified number of replica set members, including the primary. You can either specify a number or specify `majority`, which ensures the write propagates to a majority of set members.

⁵ Sharded clusters where the shards are also replica sets provide the same configuration options with regards to write and read operations.

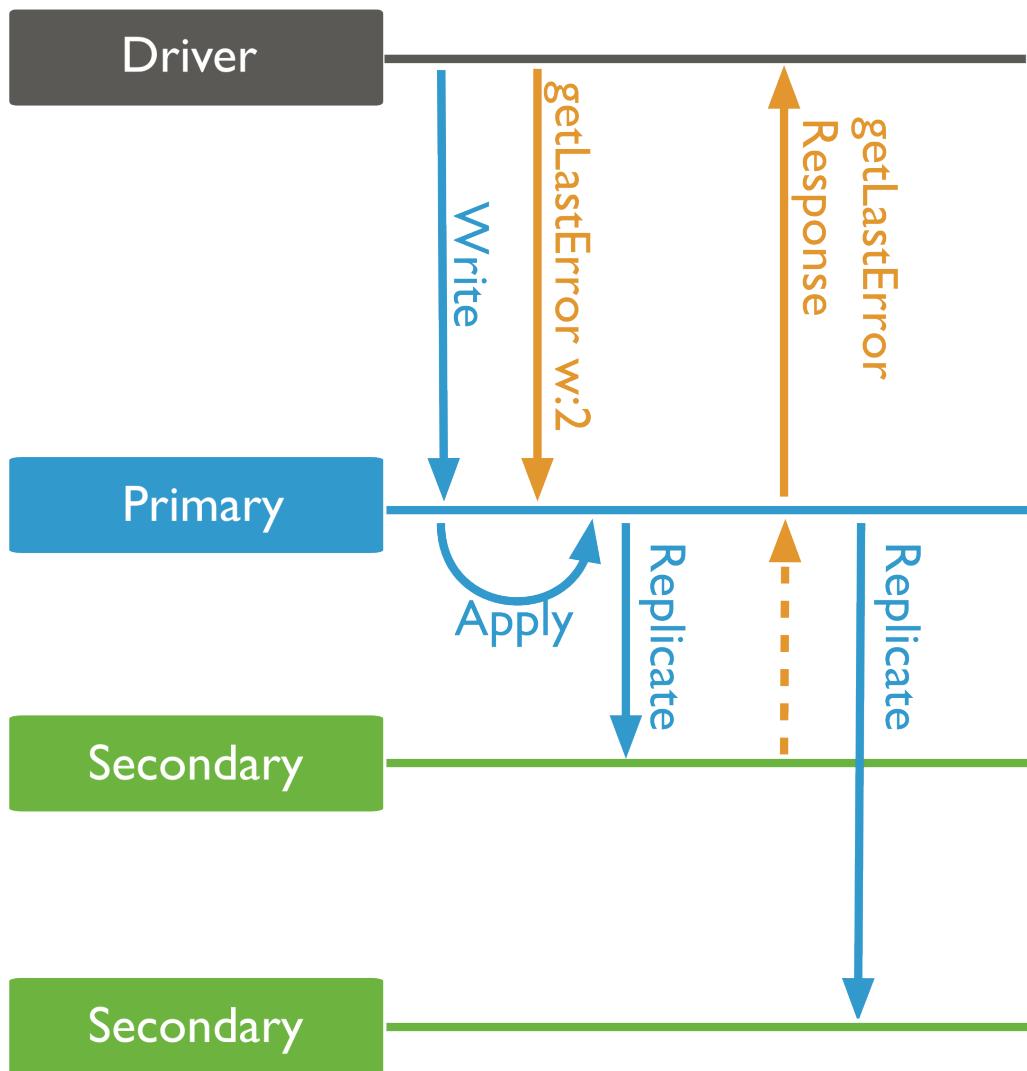


Figure 8.23: Write operation to a replica set with write concern level of `w:2` or write to the primary and at least one secondary.

If you specify a `w` value greater than the number of members that hold a copy of the data (i.e., greater than the number of non-*arbiter* members), the operation blocks until those members become available. This can cause the operation to block forever. To specify a timeout threshold for the `getLastError` operation, use the `wtimeout` argument. A `wtimeout` value of 0 means that the operation will never time out.

See *getLastError Examples* for example invocations.

Modify Default Write Concern

You can configure your own “default” `getLastError` behavior for a replica set. Use the `getLastErrorDefaults` (page 477) setting in the *replica set configuration* (page 474). The following sequence of commands creates a configuration that waits for the write operation to complete on a majority of the set members before returning:

```
cfg = rs.conf()
cfg.settings = {}
cfg.settings.getLastErrorDefaults = {w: "majority"}
rs.reconfig(cfg)
```

The `getLastErrorDefaults` (page 477) setting affects only those `getLastError` commands that have *no* other arguments.

Note: Use of insufficient write concern can lead to *rollbacks* (page 397) in the case of *replica set failover* (page 392). Always ensure that your operations have specified the required write concern for your application.

See also:

Write Concern (page 44) and *connections-write-concern*

Custom Write Concerns

You can use replica set tags to create custom write concerns using the `getLastErrorDefaults` (page 477) and `getLastErrorModes` (page 477) replica set settings.

Note: Custom write concern modes specify the field name and a number of *distinct* values for that field. By contrast, read preferences use the value of fields in the tag document to direct read operations.

In some cases, you may be able to use the same tags for read preferences and write concerns; however, you may need to create additional tags for write concerns depending on the requirements of your application.

Single Tag Write Concerns

Consider a five member replica set, where each member has one of the following tag sets:

```
{ "use": "reporting" }
{ "use": "backup" }
{ "use": "application" }
{ "use": "application" }
{ "use": "application" }
```

You could create a custom write concern mode that will ensure that applicable write operations will not return until members with two different values of the `use` tag have acknowledged the write operation. Create the mode with the following sequence of operations in the `mongo` shell:

```
cfg = rs.conf()
cfg.settings = { getLastErrorModes: { use2: { "use": 2 } } }
rs.reconfig(cfg)
```

To use this mode pass the string `use2` to the `w` option of `getLastError` as follows:

```
db.runCommand( { getLastError: 1, w: "use2" } )
```

Specific Custom Write Concerns

If you have a three member replica with the following tag sets:

```
{ "disk": "ssd" }
{ "disk": "san" }
{ "disk": "spinning" }
```

You cannot specify a custom `getLastErrorModes` (page 477) value to ensure that the write propagates to the `san` before returning. However, you may implement this write concern policy by creating the following additional tags, so that the set resembles the following:

```
{ "disk": "ssd" }
{ "disk": "san", "disk.san": "san" }
{ "disk": "spinning" }
```

Then, create a custom `getLastErrorModes` (page 477) value, as follows:

```
cfg = rs.conf()
cfg.settings = { getLastErrorModes: { san: { "disk.san": 1 } } }
rs.reconfig(cfg)
```

To use this mode pass the string `san` to the `w` option of `getLastError` as follows:

```
db.runCommand( { getLastError: 1, w: "san" } )
```

This operation will not return until a replica set member with the tag `disk.san` returns.

You may set a custom write concern mode as the default write concern mode using `getLastErrorDefaults` (page 477) replica set as in the following setting:

```
cfg = rs.conf()
cfg.settings.getLastErrorDefaults = { ssd: 1 }
rs.reconfig(cfg)
```

See also:

[Configure Replica Set Tag Sets](#) (page 446) for further information about replica set reconfiguration and tag sets.

Read Preference

Read preference describes how MongoDB clients route read operations to members of a *replica set*.

By default, an application directs its read operations to the *primary* member in a *replica set*. Reading from the primary guarantees that read operations reflect the latest version of a document. However, by distributing some or all reads to secondary members of the replica set, you can improve read throughput or reduce latency for an application that does not require fully up-to-date data.

Important: You must exercise care when specifying read preferences: modes other than `primary` (page 483) can

and will return stale data because the secondary queries will not include the most recent write operations to the replica set’s *primary*.

The following are common use cases for using non-[primary](#) (page 483) read preference modes:

- Running systems operations that do not affect the front-end application.

Issuing reads to secondaries helps distribute load and prevent operations from affecting the main workload of the primary. This can be a good choice for reporting and analytics workloads, for example.

Note: Read preferences aren’t relevant to direct connections to a single `mongod` instance. However, in order to perform read operations on a direct connection to a secondary member of a replica set, you must set a read preference, such as *secondary*.

- Providing local reads for geographically distributed applications.

If you have application servers in multiple data centers, you may consider having a [geographically distributed replica set](#) (page 392) and using a non primary read preference or the [nearest](#) (page 484). This reduces network latency by having the application server to read from a nearby secondary, rather than a distant primary.

- Maintaining availability during a failover.

Use [primaryPreferred](#) (page 483) if you want your application to do consistent reads from the primary under normal circumstances, but to allow stale reads from secondaries in an emergency. This provides a “read-only mode” for your application during a failover.

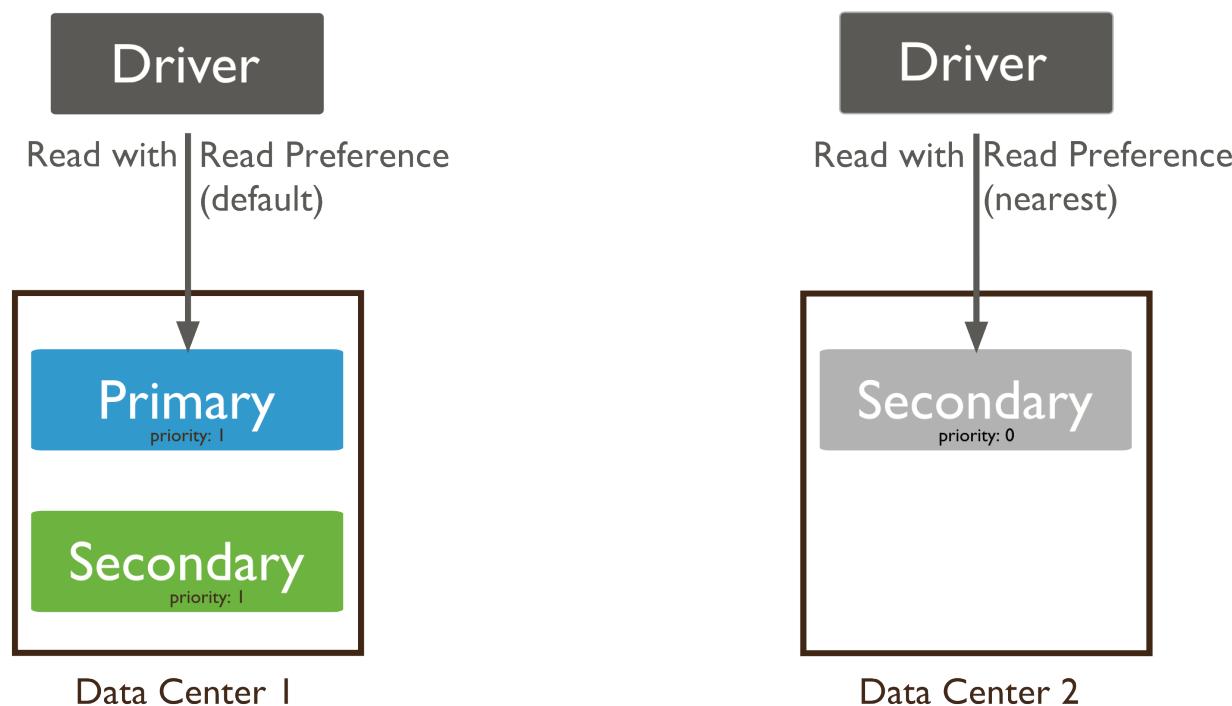


Figure 8.24: Read operations to a replica set. Default read preference routes the read to the primary. Read preference of nearest routes the read to the nearest member.

Note: In general, do not use [primary](#) (page 483) and [primaryPreferred](#) (page 483) to provide extra capacity. [Sharding](#) (page 487) increases read and write capacity by distributing read and write operations across a group of machines, and is often a better strategy for adding capacity.

See

[Read Preference Processes](#) (page 404) for more information about the internal application of read preferences.

Read Preference Modes

New in version 2.2.

Important: All read preference modes except [primary](#) (page 483) may return stale data because *secondaries* replicate operations from the primary with some delay. Ensure that your application can tolerate stale data if you choose to use a non-[primary](#) (page 483) mode.

MongoDB [drivers](#) (page 92) support five read preference modes.

Read Preference Mode	Description
primary (page 483)	Default mode. All operations read from the current replica set <i>primary</i> .
primaryPreferred (page 483)	In most situations, operations read from the <i>primary</i> but if it is unavailable, operations read from <i>secondary</i> members.
secondary (page 483)	All operations read from the <i>secondary</i> members of the replica set.
secondaryPreferred (page 484)	In most situations, operations read from <i>secondary</i> members but if no <i>secondary</i> members are available, operations read from the <i>primary</i> .
nearest (page 484)	Operations read from the <i>nearest</i> member of the <i>replica set</i> , irrespective of the member's type.

You can specify a read preference mode on connection objects, database objects, collection objects, or per-operation. The syntax for specifying the read preference mode is [specific to the driver and to the idioms of the host language](#)⁶.

Read preference modes are also available to clients connecting to a *sharded cluster* through a `mongos`. The `mongos` instance obeys specified read preferences when connecting to the *replica set* that provides each *shard* in the cluster.

In the `mongo` shell, the `readPref()` cursor method provides access to read preferences.

If read operations account for a large percentage of your application's traffic, distributing reads to secondary members can improve read throughput. However, in most cases [sharding](#) (page 492) provides better support for larger scale operations, as clusters can distribute read and write operations across a group of machines.

For more information, see [read preference background](#) (page 401) and [read preference behavior](#) (page 404). See also the [documentation for your driver](#)⁷.

Tag Sets

Tag sets allow you to specify custom [read preferences](#) (page 401) and [write concerns](#) (page 44) so that your application can target operations to specific members.

Custom read preferences and write concerns evaluate tag sets in different ways: read preferences consider the value of a tag when selecting a member to read from, while write concerns ignore the value of a tag when selecting a member *except* to consider whether or not the value is unique.

You can specify tag sets with the following read preference modes:

⁶<http://api.mongodb.org/>

⁷<http://api.mongodb.org/>

- [primaryPreferred](#) (page 483)
- [secondary](#) (page 483)
- [secondaryPreferred](#) (page 484)
- [nearest](#) (page 484)

Tags are not compatible with [primary](#) (page 483) and, in general, only apply when [selecting](#) (page 404) a [secondary](#) member of a set for a read operation. However, the [nearest](#) (page 484) read mode, when combined with a tag set will select the nearest member that matches the specified tag set, which may be a primary or secondary.

All interfaces use the same [member selection logic](#) (page 404) to choose the member to which to direct read operations, basing the choice on read preference mode and tag sets.

For information on configuring tag sets, see the [Configure Replica Set Tag Sets](#) (page 446) tutorial.

For more information on how read preference [modes](#) (page 483) interact with tag sets, see the [documentation for each read preference mode](#) (page 483).

Read Preference Processes

Changed in version 2.2.

MongoDB drivers use the following procedures to direct operations to replica sets and sharded clusters. To determine how to route their operations, applications periodically update their view of the replica set's state, identifying which members are up or down, which member is *primary*, and verifying the latency to each mongod instance.

Member Selection

Clients, by way of their drivers, and mongos instances for sharded clusters, periodically update their view of the replica set's state.

When you select non-[primary](#) (page 483) read preference, the driver will determine which member to target using the following process:

1. Assembles a list of suitable members, taking into account member type (i.e. secondary, primary, or all members).
2. Excludes members not matching the tag sets, if specified.
3. Determines which suitable member is the closest to the client in absolute terms.
4. Builds a list of members that are within a defined ping distance (in milliseconds) of the “absolute nearest” member.

Applications can configure the threshold used in this stage. The default “acceptable latency” is 15 milliseconds, which you can override in the drivers with their own `secondaryAcceptableLatencyMS` option. For mongos you can use the `--localThreshold` or `localThreshold` runtime options to set this value.

5. Selects a member from these hosts at random. The member receives the read operation.

Drivers can then associate the thread or connection with the selected member. This [request association](#) (page 404) is configurable by the application. See your [driver](#) (page 92) documentation about request association configuration and default behavior.

Request Association

Important: *Request association* is configurable by the application. See your [driver](#) (page 92) documentation about request association configuration and default behavior.

Because *secondary* members of a *replica set* may lag behind the current *primary* by different amounts, reads for *secondary* members may reflect data at different points in time. To prevent sequential reads from jumping around in time, the driver **can** associate application threads to a specific member of the set after the first read, thereby preventing reads from other members. The thread will continue to read from the same member until:

- The application performs a read with a different read preference,
- The thread terminates, or
- The client receives a socket exception, as is the case when there's a network error or when the mongod closes connections during a *failover*. This triggers a [retry](#) (page 405), which may be transparent to the application.

When using request association, if the client detects that the set has elected a new *primary*, the driver will discard all associations between threads and members.

Auto-Retry

Connections between MongoDB drivers and mongod instances in a *replica set* must balance two concerns:

1. The client should attempt to prefer current results, and any connection should read from the same member of the replica set as much as possible.
2. The client should minimize the amount of time that the database is inaccessible as the result of a connection issue, networking problem, or *failover* in a replica set.

As a result, MongoDB drivers and mongos:

- Reuse a connection to specific mongod for as long as possible after establishing a connection to that instance. This connection is *pinned* to this mongod.
- Attempt to reconnect to a new member, obeying existing [read preference modes](#) (page 483), if the connection to mongod is lost.

Reconnections are transparent to the application itself. If the connection permits reads from *secondary* members, after reconnecting, the application can receive two sequential reads returning from different secondaries. Depending on the state of the individual secondary member's replication, the documents can reflect the state of your database at different moments.

- Return an error *only* after attempting to connect to three members of the set that match the [read preference mode](#) (page 483) and [tag set](#) (page 403). If there are fewer than three members of the set, the client will error after connecting to all existing members of the set.

After this error, the driver selects a new member using the specified read preference mode. In the absence of a specified read preference, the driver uses [primary](#) (page 483).

- After detecting a failover situation,⁸ the driver attempts to refresh the state of the replica set as quickly as possible.

Read Preference in Sharded Clusters

Changed in version 2.2: Before version 2.2, mongos did not support the [read preference mode semantics](#) (page 483).

In most *sharded clusters*, each shard consists of a *replica set*. As such, read preferences are also applicable. With regard to read preference, read operations in a sharded cluster are identical to unsharded replica sets.

⁸ When a *failover* occurs, all members of the set close all client connections that produce a socket error in the driver. This behavior prevents or minimizes *rollback*.

Unlike simple replica sets, in sharded clusters, all interactions with the shards pass from the clients to the mongos instances that are actually connected to the set members. mongos is then responsible for the application of read preferences, which is transparent to applications.

There are no configuration changes required for full support of read preference modes in sharded environments, as long as the mongos is at least version 2.2. All mongos maintain their own connection pool to the replica set members. As a result:

- A request without a specified preference has [primary](#) (page 483), the default, unless, the mongos reuses an existing connection that has a different mode set.

To prevent confusion, always explicitly set your read preference mode.

- All [nearest](#) (page 484) and latency calculations reflect the connection between the mongos and the mongod instances, not the client and the mongod instances.

This produces the desired result, because all results must pass through the mongos before returning to the client.

8.2.5 Replication Processes

Members of a *replica set* replicate data continuously. First, a member uses *initial sync* to capture the data set. Then the member continuously records and applies every operation that modifies the data set. Every member records operations in its [oplog](#) (page 406), which is a *capped collection*.

Replica Set Oplog (page 406) The oplog records all operations that modify the data in the replica set.

Replica Set Data Synchronization (page 407) Secondaries must replicate all changes accepted by the primary. This process is the basis of replica set operations.

Replica Set Oplog

The *oplog* (operations log) is a special *capped collection* that keeps a rolling record of all operations that modify the data stored in your databases. MongoDB applies database operations on the *primary* and then records the operations on the primary's oplog. The *secondary* members then copy and apply these operations in an asynchronous process. All replica set members contain a copy of the oplog, allowing them to maintain the current state of the database.

To facilitate replication, all replica set members send heartbeats (pings) to all other members. Any member can import oplog entries from any other member.

Whether applied once or multiple times to the target dataset, each operation in the oplog produces the same results, i.e. each operation in the oplog is *idempotent*. For proper replication operations, entries in the oplog must be idempotent:

- initial sync
- post-rollback catch-up
- sharding chunk migrations

Oplog Size

When you start a replica set member for the first time, MongoDB creates an oplog of a default size. The size depends on the architectural details of your operating system.

In most cases, the default oplog size is sufficient. For example, if an oplog is 5% of free disk space and fills up in 24 hours of operations, then secondaries can stop copying entries from the oplog for up to 24 hours without becoming stale. However, most replica sets have much lower operation volumes, and their oplogs can hold much higher numbers of operations.

Before `mongod` creates an oplog, you can specify its size with the `oplogSize` option. However, after you have started a replica set member for the first time, you can only change the size of the oplog using the [Change the Size of the Oplog](#) (page 441) procedure.

By default, the size of the oplog is as follows:

- For 64-bit Linux, Solaris, FreeBSD, and Windows systems, MongoDB allocates 5% of the available free disk space to the oplog. If this amount is smaller than a gigabyte, then MongoDB allocates 1 gigabyte of space.
- For 64-bit OS X systems, MongoDB allocates 183 megabytes of space to the oplog.
- For 32-bit systems, MongoDB allocates about 48 megabytes of space to the oplog.

Workloads that Might Require a Larger Oplog Size

If you can predict your replica set's workload to resemble one of the following patterns, then you might want to create an oplog that is larger than the default. Conversely, if your application predominantly performs reads and writes only a small amount of data, you will oplog may be sufficient.

The following workloads might require a larger oplog size.

Updates to Multiple Documents at Once The oplog must translate multi-updates into individual operations in order to maintain *idempotency*. This can use a great deal of oplog space without a corresponding increase in data size or disk use.

Deletions Equal the Same Amount of Data as Inserts If you delete roughly the same amount of data as you insert, the database will not grow significantly in disk use, but the size of the operation log can be quite large.

Significant Number of In-Place Updates If a significant portion of the workload is in-place updates, the database records a large number of operations but does not change the quantity of data on disk.

Oplog Status

To view oplog status, including the size and the time range of operations, issue the `db.printReplicationInfo()` method. For more information on oplog status, see [Check the Size of the Oplog](#) (page 460).

Under various exceptional situations, updates to a *secondary*'s oplog might lag behind the desired performance time. Use `db.getReplicationInfo()` from a secondary member and the `replication` status output to assess the current state of replication and determine if there is any unintended replication delay.

See [Replication Lag](#) (page 458) for more information.

Replica Set Data Synchronization

In order to maintain up-to-date copies of the shared data set, members of a replica set sync or replicate data from other members. MongoDB uses two forms of data synchronization: [initial sync](#) (page 408) to populate new members with the full data set, and replication to apply ongoing changes to the entire data set.

Initial Sync

Initial sync copies all the data from one member of the replica set to another member. A member uses initial sync when the member has no data, such as when the member is new, or when the member has data but is missing a history of the set's replication.

When you perform an initial sync, MongoDB does the following:

1. Clones all databases. To clone, the `mongod` queries every collection in each source database and inserts all data into its own copies of these collections.
2. Applies all changes to the data set. Using the oplog from the source, the `mongod` updates its data set to reflect the current state of the replica set.
3. Builds all indexes on all collections.

When the `mongod` finishes building all index builds, the member can transition to a normal state, i.e. *secondary*.

To perform an initial sync, see [Resync a Member of a Replica Set](#) (page 445).

Replication

Replica set members replicate data continuously after the initial sync. This process keeps the members up to date with all changes to the replica set's data. In most cases, secondaries synchronize from the primary. Secondaries may automatically change their *sync targets* if needed based on changes in the ping time and state of other members' replication.

For a member to sync from another, the `buildIndexes` (page 475) setting for both members must have the same value/ `buildIndexes` (page 475) must be either `true` or `false` for both members.

Beginning in version 2.2, secondaries avoid syncing from *delayed members* (page 383) and *hidden members* (page 383).

Consistency and Durability

In a replica set, only the primary can accept write operations. Writing only to the primary provides *strict consistency* among members.

Journaling provides single-instance write durability. Without journaling, if a MongoDB instance terminates ungracefully, you should assume that the database is in a corrupt or inconsistent state.

Multithreaded Replication

MongoDB applies write operations in batches using multiple threads to improve concurrency. MongoDB groups batches by namespace and applies operations using a group of threads, but always applies the write operations to a namespace in order.

While applying a batch, MongoDB blocks all reads. As a result, secondaries can never return data that reflects a state that never existed on the primary.

Pre-Fetching Indexes to Improve Replication Throughput

To help improve the performance of applying oplog entries, MongoDB fetches memory pages that hold affected data and indexes. This *pre-fetch* stage minimizes the amount of time MongoDB holds the write lock while applying oplog entries. By default, secondaries will pre-fetch all *Indexes* (page 309).

Optionally, you can disable all pre-fetching or only pre-fetch the index on the `_id` field. See the `replIndexPrefetch` setting for more information.

8.2.6 Master Slave Replication

Important: [Replica sets](#) (page 377) replace *master-slave* replication for most use cases. If possible, use replica sets rather than master-slave replication for all new production deployments. This documentation remains to support legacy deployments and for archival purposes only.

In addition to providing all the functionality of master-slave deployments, replica sets are also more robust for production use. Master-slave replication preceded replica sets and made it possible have a large number of non-master (i.e. slave) nodes, as well as to restrict replicated operations to only a single database; however, master-slave replication provides less redundancy and does not automate failover. See [Deploy Master-Slave Equivalent using Replica Sets](#) (page 411) for a replica set configuration that is equivalent to master-slave replication. If you wish to convert an existing master-slave deployment to a replica set, see [Convert a Master-Slave Deployment to a Replica Set](#) (page 412).

Fundamental Operations

Initial Deployment

To configure a *master-slave* deployment, start two `mongod` instances: one in `master` mode, and the other in `slave` mode.

To start a `mongod` instance in `master` mode, invoke `mongod` as follows:

```
mongod --master --dbpath /data/masterdb/
```

With the `--master` option, the `mongod` will create a `local.oplog.$main` (page 481) collection, which the “operation log” that queues operations that the slaves will apply to replicate operations from the master. The `--dbpath` is optional.

To start a `mongod` instance in `slave` mode, invoke `mongod` as follows:

```
mongod --slave --source <masterhostname><:<port>> --dbpath /data/slavedb/
```

Specify the hostname and port of the master instance to the `--source` argument. The `--dbpath` is optional.

For slave instances, MongoDB stores data about the source server in the `local.sources` (page 481) collection.

Configuration Options for Master-Slave Deployments

As an alternative to specifying the `--source` run-time option, can add a document to `local.sources` (page 481) specifying the master instance, as in the following operation in the `mongo` shell:

```
1 use local
2 db.sources.find()
3 db.sources.insert( { host: <masterhostname> <,only: databasename> } );
```

In line 1, you switch context to the `local` database. In line 2, the `find()` operation should return no documents, to ensure that there are no documents in the `sources` collection. Finally, line 3 uses `db.collection.insert()` to insert the source document into the `local.sources` (page 481) collection. The model of the `local.sources` (page 481) document is as follows:

host

The host field specifies the master mongod instance, and holds a resolvable hostname, i.e. IP address, or a name from a host file, or preferably a fully qualified domain name.

You can append <:port> to the host name if the mongod is not running on the default 27017 port.

only

Optional. Specify a name of a database. When specified, MongoDB will only replicate the indicated database.

Operational Considerations for Replication with Master Slave Deployments

Master instances store operations in an *oplog* which is a *capped collection* (page 158). As a result, if a slave falls too far behind the state of the master, it cannot “catchup” and must re-sync from scratch. Slave may become out of sync with a master if:

- The slave falls far behind the data updates available from that master.
- The slave stops (i.e. shuts down) and restarts later after the master has overwritten the relevant operations from the master.

When slaves, are out of sync, replication stops. Administrators must intervene manually to restart replication. Use the `resync` command. Alternatively, the `--autoresync` allows a slave to restart replication automatically, after ten second pause, when the slave falls out of sync with the master. With `--autoresync` specified, the slave will only attempt to re-sync once in a ten minute period.

To prevent these situations you should specify a larger oplog when you start the master instance, by adding the `--oplogSize` option when starting mongod. If you do not specify `--oplogSize`, mongod will allocate 5% of available disk space on start up to the oplog, with a minimum of 1GB for 64bit machines and 50MB for 32bit machines.

Run time Master-Slave Configuration

MongoDB provides a number of run time configuration options for mongod instances in *master-slave* deployments. You can specify these options in *configuration files* (page 143) or on the command-line. See documentation of the following:

- For *master* nodes:
 - `master`
 - `slave`
- For *slave* nodes:
 - `source`
 - `only`
 - `slaveDelay`

Also consider the *Master-Slave Replication Command Line Options* for related options.

Diagnostics

On a *master* instance, issue the following operation in the mongo shell to return replication status from the perspective of the master:

```
db.printReplicationInfo()
```

On a *slave* instance, use the following operation in the `mongo` shell to return the replication status from the perspective of the slave:

```
db.printSlaveReplicationInfo()
```

Use the `serverStatus` as in the following operation, to return status of the replication:

```
db.serverStatus()
```

See *server status repl fields* for documentation of the relevant section of output.

Security

When running with `auth` enabled, in *master-slave* deployments configure a `keyFile` so that slave `mongod` instances can authenticate and communicate with the master `mongod` instance.

To enable authentication and configure the `keyFile` add the following option to your configuration file:

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

Note: You may chose to set these run-time configuration options using the `--keyFile` option on the command line.

Setting `keyFile` enables authentication and specifies a key file for the `mongod` instances to use when authenticating to each other. The content of the key file is arbitrary but must be the same on all members of the deployment can connect to each other.

The key file must be less one kilobyte in size and may only contain characters in the base64 set. The key file must not have group or “world” permissions on UNIX systems. Use the following command to use the OpenSSL package to generate “random” content for use in a key file:

```
openssl rand -base64 741
```

See also:

[Security](#) (page 233) for more information about security in MongoDB

Ongoing Administration and Operation of Master-Slave Deployments

Deploy Master-Slave Equivalent using Replica Sets

If you want a replication configuration that resembles *master-slave* replication, using *replica sets* replica sets, consider the following replica configuration document. In this deployment hosts `<master>` and `<slave>`⁹ provide replication that is roughly equivalent to a two-instance master-slave deployment:

```
{
  _id : 'setName',
  members : [
    { _id : 0, host : "<master>", priority : 1 },
    { _id : 1, host : "<slave>", priority : 0, votes : 0 }
  ]
}
```

See [Replica Set Configuration](#) (page 474) for more information about replica set configurations.

⁹ In replica set configurations, the `host` (page 475) field must hold a resolvable hostname.

Convert a Master-Slave Deployment to a Replica Set

To convert a master-slave deployment to a replica set, restart the current master as a one-member replica set. Then remove the data directors from previous secondaries and add them as new secondaries to the new replica set.

1. To confirm that the current instance is master, run:

```
db.isMaster()
```

This should return a document that resembles the following:

```
{  
    "ismaster" : true,  
    "maxBsonObjectSize" : 16777216,  
    "maxMessageSizeBytes" : 48000000,  
    "localTime" : ISODate("2013-07-08T20:15:13.664Z"),  
    "ok" : 1  
}
```

2. Shut down the mongod processes on the master and all slave(s), using the following command while connected to each instance:

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

3. Back up your /data/db directories, in case you need to revert to the master-slave deployment.

4. Start the former master with the `--replSet` option, as in the following:

```
mongod --replSet <setname>
```

5. Connect to the mongod with the mongo shell, and initiate the replica set with the following command:

```
rs.initiate()
```

When the command returns, you will have successfully deployed a one-member replica set. You can check the status of your replica set at any time by running the following command:

```
rs.status()
```

You can now follow the [convert a standalone to a replica set](#) (page 428) tutorial to deploy your replica set, picking up from the [Expand the Replica Set](#) (page 428) section.

Failing over to a Slave (Promotion)

To permanently failover from a unavailable or damaged *master* (A in the following example) to a *slave* (B):

1. Shut down A.
2. Stop mongod on B.
3. Back up and move all data files that begin with local on B from the dbpath.

Warning: Removing local.* is irrevocable and cannot be undone. Perform this step with extreme caution.

4. Restart mongod on B with the `--master` option.

Note: This is a one time operation, and is not reversible. A cannot become a slave of B until it completes a full resync.

Inverting Master and Slave

If you have a *master* (A) and a *slave* (B) and you would like to reverse their roles, follow this procedure. The procedure assumes A is healthy, up-to-date and available.

If A is not healthy but the hardware is okay (power outage, server crash, etc.), skip steps 1 and 2 and in step 8 replace all of A's files with B's files in step 8.

If A is not healthy and the hardware is not okay, replace A with a new machine. Also follow the instructions in the previous paragraph.

To invert the master and slave in a deployment:

1. Halt writes on A using the `fsync` command.
2. Make sure B is up to date with the state of A.
3. Shut down B.
4. Back up and move all data files that begin with `local` on B from the `dbpath` to remove the existing `local.sources` data.

Warning: Removing `local.*` is irrevocable and cannot be undone. Perform this step with extreme caution.

5. Start B with the `--master` option.
6. Do a write on B, which primes the `oplog` to provide a new sync start point.
7. Shut down B. B will now have a new set of data files that start with `local`.
8. Shut down A and replace all files in the `dbpath` of A that start with `local` with a copy of the files in the `dbpath` of B that begin with `local`.

Considering compressing the `local` files from B while you copy them, as they may be quite large.

9. Start B with the `--master` option.
10. Start A with all the usual slave options, but include `fastsync`.

Creating a Slave from an Existing Master's Disk Image

If you can stop write operations to the *master* for an indefinite period, you can copy the data files from the master to the new *slave* and then start the slave with `--fastsync`.

Warning: Be careful with `--fastsync`. If the data on both instances is identical, a discrepancy will exist forever.

`fastsync` is a way to start a slave by starting with an existing master disk image/backup. This option declares that the administrator guarantees the image is correct and completely up-to-date with that of the master. If you have a full and complete copy of data from a master you can use this option to avoid a full synchronization upon starting the slave.

Creating a Slave from an Existing Slave's Disk Image

You can just copy the other *slave*'s data file snapshot without any special options. Only take data snapshots when a `mongod` process is down or locked using `db.fsyncLock()`.

Resyncing a Slave that is too Stale to Recover

Slaves asynchronously apply write operations from the *master* that the slaves poll from the master's *oplog*. The oplog is finite in length, and if a slave is too far behind, a full resync will be necessary. To resync the slave, connect to a slave using the mongo and issue the `resync` command:

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

This forces a full resync of all data (which will be very slow on a large database). You can achieve the same effect by stopping mongod on the slave, deleting the entire content of the dbpath on the slave, and restarting the mongod.

Slave Chaining

Slaves cannot be “chained.” They must all connect to the *master* directly.

If a slave attempts “slave from” another slave you will see the following line in the mongod log of the shell:

```
assertion 13051 tailable cursor requested on non capped collection ns:local.oplog.$main
```

Correcting a Slave’s Source

To change a *slave*’s source, manually modify the slave’s `local.sources` (page 481) collection.

Example

Consider the following: If you accidentally set an incorrect hostname for the slave’s source, as in the following example:

```
mongod --slave --source prod.mississippi
```

You can correct this, by restarting the slave without the `--slave` and `--source` arguments:

```
mongod
```

Connect to this mongod instance using the mongo shell and update the `local.sources` (page 481) collection, with the following operation sequence:

```
use local
```

```
db.sources.update( { host : "prod.mississippi" },
                    { $set : { host : "prod.mississippi.example.net" } } )
```

Restart the slave with the correct command line arguments or with no `--source` option. After configuring `local.sources` (page 481) the first time, the `--source` will have no subsequent effect. Therefore, both of the following invocations are correct:

```
mongod --slave --source prod.mississippi.example.net
```

or

```
mongod --slave
```

The slave now polls data from the correct *master*.

8.3 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 \(page 415\)](#) Instructions for deploying replica sets, as well as adding and removing members from an existing replica set.

[Deploy a Replica Set \(page 416\)](#) Configure a three-member replica set for either a production system.

[Convert a Standalone to a Replica Set \(page 428\)](#) Convert an existing standalone `mongod` instance into a three-member replica set.

[Add Members to a Replica Set \(page 429\)](#) Add a new member to an existing replica set.

[Remove Members from Replica Set \(page 431\)](#) Remove a member from a replica set.

[Member Configuration Tutorials \(page 433\)](#) Tutorials that describe the process for configuring replica set members.

[Adjust Priority for Replica Set Member \(page 434\)](#) Change the precedence given to a replica set members in an election for primary.

[Prevent Secondary from Becoming Primary \(page 434\)](#) Make a secondary member ineligible for election as primary.

[Configure a Hidden Replica Set Member \(page 436\)](#) Configure a secondary member to be invisible to applications in order to support significantly different usage, such as a dedicated backups.

[Replica Set Maintenance Tutorials \(page 440\)](#) Procedures and tasks for common operations on active replica set deployments.

[Change the Size of the Oplog \(page 441\)](#) 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 \(page 445\)](#) 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.

[Change the Size of the Oplog \(page 441\)](#) Increase the size of the *oplog* which logs operations. In most cases, the default oplog size is sufficient.

[Force a Member to Become Primary \(page 443\)](#) Force a replica set member to become primary.

[Change Hostnames in a Replica Set \(page 453\)](#) Update the replica set configuration to reflect changes in members' hostnames.

[Troubleshoot Replica Sets \(page 457\)](#) Describes common issues and operational challenges for replica sets. For additional diagnostic information, see [FAQ: MongoDB Diagnostics \(page 606\)](#).

8.3.1 Replica Set Deployment Tutorials

The following tutorials provide information in deploying replica sets.

[Deploy a Replica Set \(page 416\)](#) Configure a three-member replica set for either a production system.

[Deploy a Replica Set for Testing and Development \(page 419\)](#) Configure a three-member replica set for either a development and testing systems.

[Deploy a Geographically Redundant Replica Set \(page 421\)](#) 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 \(page 427\)](#) Add an arbiter give a replica set an odd number of voting members to prevent election ties.

[Convert a Standalone to a Replica Set \(page 428\)](#) Convert an existing standalone mongod instance into a three-member replica set.

[Add Members to a Replica Set \(page 429\)](#) Add a new member to an existing replica set.

[Remove Members from Replica Set \(page 431\)](#) Remove a member from a replica set.

[Replace a Replica Set Member \(page 433\)](#) Update the replica set configuration when the hostname of a member's corresponding mongod instance has changed.

Deploy a Replica Set

This tutorial describes how to create a three-member *replica set* from three existing mongod instances.

If you wish to deploy a replica set from a single MongoDB instance, see [Convert a Standalone to a Replica Set \(page 428\)](#). For more information on replica set deployments, see the [Replication \(page 373\)](#) and [Replica Set Deployment Architectures \(page 386\)](#) 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](#) (page 393) will proceed smoothly. For more about designing replica sets, see [the Replication overview \(page 373\)](#).

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 \(page 3\)](#).

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 \(page 459\)](#).

Procedure

- Each member of the replica set resides on its own machine and all of the MongoDB processes bind to port 27017 (the standard MongoDB port).
- Each member of the replica set must be accessible by way of resolvable DNS or hostnames, as in the following scheme:
 - mongodb0.example.net

- mongodb1.example.net
- mongodb2.example.net
- mongodbn.example.net

You will need to *either* configure your DNS names appropriately, *or* set up your systems' /etc/hosts file to reflect this configuration.

- Ensure that network traffic can pass between all members 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 authentication using auth and keyFile, so that only servers and processes with authentication can connect to the replica set.
 - Configure networking and firewall rules so that only traffic (incoming and outgoing packets) on the default MongoDB port (e.g. 27017) from *within* your deployment is permitted.

For more information on security and firewalls, see [Inter-Process Authentication](#) (page 236).

- You must specify the run time configuration on each system in a configuration file stored in /etc/mongodb.conf or a related location. *Do not* specify the set's configuration in the mongo shell.

Use the following configuration for each of your MongoDB instances. You should set values that are appropriate for your systems, as needed:

```
port = 27017
bind_ip = 10.8.0.10
dbpath = /srv/mongodb/
fork = true
replSet = rs0
```

The dbpath indicates where you want mongod to store data files. The dbpath must exist before you start mongod. If it does not exist, create the directory and ensure mongod has permission to read and write data to this path. For more information on permissions, see the [security operations documentation](#) (page 234).

Modifying bind_ip ensures that mongod will only listen for connections from applications on the configured address.

For more information about the run time options used above and other configuration options, see <http://docs.mongodb.org/manual/reference/configuration-options>.

To deploy a production replica set:

1. Start a mongod instance on each system that will be part of your replica set. Specify the same replica set name on each instance. For additional mongod configuration options specific to replica sets, see [cli-mongod-replica-set](#).

Important: 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.

If you use a configuration file, then start each mongod instance with a command that resembles the following:

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

Change `/etc/mongodb.conf` to the location of your configuration file.

Note: You will likely want to use and configure a *control script* to manage this process in production deployments. Control scripts are beyond the scope of this document.

2. Open a mongo shell connected to **one** of the hosts by issuing the following command:

```
mongo
```

3. Use `rs.initiate()` to initiate a replica set consisting of the current member and using the default configuration, as follows:

```
rs.initiate()
```

4. Display the current *replica configuration* (page 474):

```
rs.conf()
```

The replica set configuration object resembles the following

```
{
  "_id" : "rs0",
  "version" : 4,
  "members" : [
    {
      "_id" : 1,
      "host" : "mongodb0.example.net:27017"
    }
  ]
}
```

1. In the mongo shell connected to the *primary*, add the remaining members to the replica set using `rs.add()` in the mongo shell on the current primary (in this example, `mongodb0.example.net`). The commands should resemble the following:

```
rs.add("mongodb1.example.net")
rs.add("mongodb2.example.net")
```

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()`

Refer to *Replica Set Read and Write Semantics* (page 398) for a detailed explanation of read and write semantics in MongoDB.

Deploy a Replica Set for Testing and Development

Note: This tutorial provides instructions for deploying a replica set in a development or test environment. For a production deployment, refer to the [Deploy a Replica Set](#) (page 416) tutorial.

This tutorial describes how to create a three-member *replica set* from three existing `mongod` instances.

If you wish to deploy a replica set from a single MongoDB instance, see [Convert a Standalone to a Replica Set](#) (page 428). For more information on replica set deployments, see the [Replication](#) (page 373) and [Replica Set Deployment Architectures](#) (page 386) 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](#) (page 393) will proceed smoothly. For more about designing replica sets, see [the Replication overview](#) (page 373).

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](#) (page 3).

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 459).

Procedure

Important: These instructions should only be used for test or development deployments.

The examples in this procedure create a new replica set named `rs0`.

Important: 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`.

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 <http://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* (page 474) 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"
        }
    ]
}
```

- 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](#)¹⁰ as an example of a basic automatically-configured replica set.

Refer to [Replica Set Read and Write Semantics](#) (page 398) for a detailed explanation of read and write semantics in MongoDB.

Deploy a Geographically Redundant Replica Set

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 [Replication](#) (page 373) and [Replica Set Deployment Architectures](#) (page 386). For related tutorials, see [Deploy a Replica Set](#) (page 416) and [Add Members to a Replica Set](#) (page 429).

Overview

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.

Requirements

In general, the requirements for any geographically redundant replica set are as follows:

- Ensure that a majority of the [voting members](#) (page 396) are within a primary facility, “Site A”. This includes [priority 0 members](#) (page 382) and [arbiters](#) (page 385). Deploy other members in secondary facilities, “Site B”, “Site C”, etc., to provide additional copies of the data. See [Determine the Distribution of Members](#) (page 387) 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](#) (page 385) 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.)

¹⁰<https://github.com/mongodb/mongo-snippets/blob/master/replication/simple-setup.py>

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](#) (page 3).

Procedures

General Considerations

- Each member of the replica set resides on its own machine and all of the MongoDB processes bind to port 27017 (the standard MongoDB port).
- Each member of the replica set must be accessible by way of resolvable DNS or hostnames, as in the following scheme:
 - `mongodb0.example.net`
 - `mongodb1.example.net`
 - `mongodb2.example.net`
 - `mongodbn.example.net`

You will need to *either* configure your DNS names appropriately, *or* set up your systems' `/etc/hosts` file to reflect this configuration.

- Ensure that network traffic can pass between all members 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 authentication using `auth` and `keyFile`, so that only servers and processes with authentication can connect to the replica set.
 - Configure networking and firewall rules so that only traffic (incoming and outgoing packets) on the default MongoDB port (e.g. 27017) from *within* your deployment is permitted.

For more information on security and firewalls, see [Inter-Process Authentication](#) (page 236).

- You must specify the run time configuration on each system in a configuration file stored in `/etc/mongodb.conf` or a related location. *Do not* specify the set's configuration in the `mongo` shell.

Use the following configuration for each of your MongoDB instances. You should set values that are appropriate for your systems, as needed:

```
port = 27017  
  
bind_ip = 10.8.0.10  
  
dbpath = /srv/mongodb/  
  
fork = true  
  
replSet = rs0
```

The dbpath indicates where you want mongod to store data files. The dbpath must exist before you start mongod. If it does not exist, create the directory and ensure mongod has permission to read and write data to this path. For more information on permissions, see the [security operations documentation](#) (page 234).

Modifying bind_ip ensures that mongod will only listen for connections from applications on the configured address.

For more information about the run time options used above and other configuration options, see <http://docs.mongodb.org/manual/reference/configuration-options>.

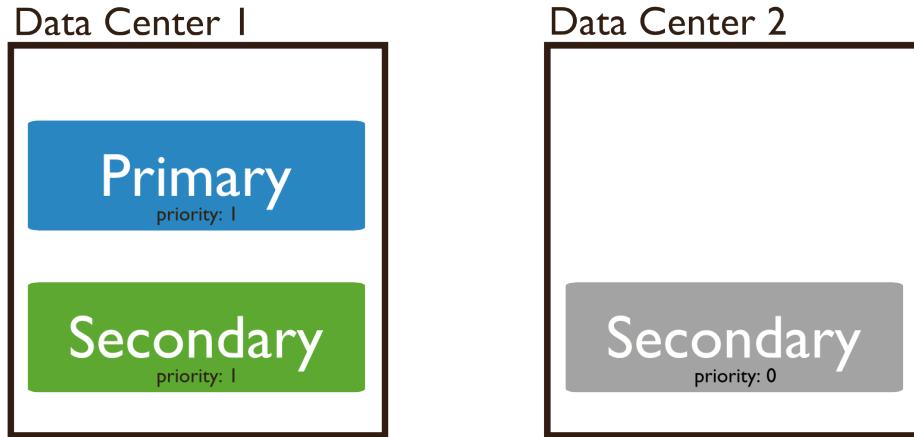


Figure 8.25: Diagram of a 3 member replica set distributed across two data centers. Replica set includes a priority 0 member.

Deploy a Geographically Redundant Three-Member Replica Set

1. Start a mongod instance on each system that will be part of your replica set. Specify the same replica set name on each instance. For additional mongod configuration options specific to replica sets, see [cli-mongod-replica-set](#).

Important: 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.

If you use a configuration file, then start each mongod instance with a command that resembles the following:

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

Change /etc/mongodb.conf to the location of your configuration file.

Note: You will likely want to use and configure a *control script* to manage this process in production deployments. Control scripts are beyond the scope of this document.

2. Open a mongo shell connected to **one** of the hosts by issuing the following command:

```
mongo
```

3. Use rs.initiate() to initiate a replica set consisting of the current member and using the default configuration, as follows:

```
rs.initiate()
```

4. Display the current *replica configuration* (page 474):

```
rs.conf()
```

The replica set configuration object resembles the following

```
{
  "_id" : "rs0",
  "version" : 4,
  "members" : [
    {
      "_id" : 1,
      "host" : "mongodb0.example.net:27017"
    }
  ]
}
```

1. In the mongo shell connected to the *primary*, add the remaining members to the replica set using `rs.add()` in the mongo shell on the current primary (in this example, `mongodb0.example.net`). The commands should resemble the following:

```
rs.add("mongodb1.example.net")
rs.add("mongodb2.example.net")
```

When complete, you should have a fully functional replica set. The new replica set will elect a *primary*.

6. Make sure that you have configured the member located in Site B (in this example, `mongodb2.example.net`) as a *priority 0 member* (page 382):

- (a) Issue the following command to determine the `members` (page 474) array position for the member:

```
rs.conf()
```

- (b) In the `members` (page 474) array, save the position of the member whose priority you wish to change. The example in the next step assumes this value is 2, for the third item in the list. You must record *array position*, not `_id`, as these ordinals will be different if you remove a member.

- (c) In the mongo shell connected to the replica set's primary, issue a command sequence similar to the following:

```
cfg = rs.conf()
cfg.members[2].priority = 0
rs.reconfig(cfg)
```

When the operations return, `mongodb2.example.net` has a priority of 0. It cannot become primary.

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.

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()`

Refer to [Replica Set Read and Write Semantics](#) (page 398) for a detailed explanation of read and write semantics in MongoDB.

Deploy a Geographically Redundant Four-Member Replica Set A geographically redundant four-member deployment has two additional considerations:

- One host (e.g. `mongodb3.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* (page 382) in Site B, and an arbiter in Site A.
 - Two members in Site A, two *priority 0 members* (page 382) 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:

1. Start a `mongod` instance on each system that will be part of your replica set. Specify the same replica set name on each instance. For additional `mongod` configuration options specific to replica sets, see [cli-mongod-replica-set](#).

Important: 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.

If you use a configuration file, then start each `mongod` instance with a command that resembles the following:

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

Change `/etc/mongodb.conf` to the location of your configuration file.

Note: You will likely want to use and configure a *control script* to manage this process in production deployments. Control scripts are beyond the scope of this document.

2. Open a `mongo` shell connected to **one** of the hosts by issuing the following command:

```
mongo
```

3. Use `rs.initiate()` to initiate a replica set consisting of the current member and using the default configuration, as follows:

```
rs.initiate()
```

4. Display the current *replica configuration* (page 474):

```
rs.conf()
```

The replica set configuration object resembles the following

```
{  
    "_id" : "rs0",  
    "version" : 4,  
    "members" : [  
        {  
            "_id" : 1,  
            "host" : "mongodb0.example.net:27017"  
        }  
    ]  
}
```

5. Add the remaining members to the replica set using `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*.

6. In the same shell session, issue the following command to add the arbiter (e.g. `mongodb4.example.net`):

```
rs.addArb("mongodb4.example.net")
```

7. Make sure that you have configured each member located outside of Site A (e.g. `mongodb3.example.net`) as a *priority 0 member* (page 382):

- (a) Issue the following command to determine the `members` (page 474) array position for the member:

```
rs.conf()
```

- (b) In the `members` (page 474) array, save the position of the member whose priority you wish to change. The example in the next step assumes this value is 2, for the third item in the list. You must record *array position*, not `_id`, as these ordinals will be different if you remove a member.

- (c) In the mongo shell connected to the replica set's primary, issue a command sequence similar to the following:

```
cfg = rs.conf()  
cfg.members[2].priority = 0  
rs.reconfig(cfg)
```

When the operations return, `mongodb2.example.net` has a priority of 0. It cannot become primary.

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.

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()`

Refer to [Replica Set Read and Write Semantics](#) (page 398) for a detailed explanation of read and write semantics in MongoDB.

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 425)). Ensure that the 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 423). 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

Arbiters are `mongod` instances that are part of *replica set* but do not hold data. Arbiters participate in *elections* (page 393) 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, monitoring host.

Important: Do not run an arbiter on the same system as a member of the replica set.

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

- [Procedure](#) (page 428)
 - [Expand the Replica Set](#) (page 428)
 - [Sharding Considerations](#) (page 429)

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](#) (page 3).

To deploy a replica set without using a pre-existing mongod instance, see [Deploy a Replica Set](#) (page 416).

Procedure

1. Shut down the *standalone* mongod instance.
2. Restart the instance. Use the `--rep1Set` 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 --rep1Set rs0
```

Important: 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 <http://docs.mongodb.org/manual/reference/configuration/> 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](#) (page 3) 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 429).

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:

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

Overview

This tutorial explains how to add an additional member to an existing *replica set*. For background on replication deployment patterns, see the [Replica Set Deployment Architectures](#) (page 386) document.

Maximum Voting Members A replica set can have a maximum of seven *voting members* (page 393). To add a member to a replica set that already has seven votes, you must either add the member as a *non-voting member* (page 396) or remove a vote from an *existing member* (page 476).

Control Scripts In production deployments you can configure a *control 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 consistent copy of the database from a member of the same replica set. See [Backup and Restore with Filesystem Snapshots](#) (page 188) document for more information.

Important: Always use filesystem snapshots to create a copy 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](#) (page 3) and the [Deploy a Replica Set](#) (page 416) 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* (page 406). 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 445).

Use `db.printReplicationInfo()` to check the current state of replica set members with regards to the oplog.

For background on replication deployment patterns, see the *Replica Set Deployment Architectures* (page 386) 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 `mongo.conf` configuration file, and start the `mongod` with the following command:

```
mongod --config /etc/mongodb.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* (page 474):

```
rs.conf()
```

To view replica set status, issue the `rs.status()` method. For a description of the status fields, see <http://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` (page 474) document. The document must be in the form of a `local.system.replset.members` (page 474) document. These documents define a replica set member in the same form as the *replica set configuration document* (page 474).

Important: Specify a value for the `_id` field of the `members` (page 474) document. MongoDB does not automatically populate the `_id` field in this case. Finally, the `members` (page 474) 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` (page 475) of `mongodb3.example.net:27017`.
- a `priority` (page 476) value within the replica set of 0.
- a configuration as `hidden` (page 475),

Issue the following:

```
rs.add({_id: 1, host: "mongodb3.example.net:27017", priority: 0, hidden: true})
```

Remove Members from Replica Set

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 `DBCClientCursor::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* (page 474), 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

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` (page 475) field. The value of `_id` (page 474) field will not change when you reconfigure the set.

See [Replica Set Configuration](#) (page 474) and `rs.reconfig()` for more information.

Note: Any replica set configuration change can trigger the current *primary* to step down, which forces an *election* (page 393). 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)
```

8.3.2 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 434) Change the precedence given to a replica set members in an election for primary.

[Prevent Secondary from Becoming Primary](#) (page 434) Make a secondary member ineligible for election as primary.

[Configure a Hidden Replica Set Member](#) (page 436) 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 437) 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 438) 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 438) Convert a secondary to an arbiter.

Adjust Priority for Replica Set Member

To change the value of the [priority](#) (page 476) in the replica set configuration, use the following sequence of commands in the mongo shell:

```
cfg = rs.conf()
cfg.members[0].priority = 0.5
cfg.members[1].priority = 2
cfg.members[2].priority = 2
rs.reconfig(cfg)
```

The first operation uses `rs.conf()` to set the local variable `cfg` to the contents of the current replica set configuration, which is a *document*. The next three operations change the [priority](#) (page 476) value in the `cfg` document for the first three members configured in the [members](#) (page 474) array. The final operation calls `rs.reconfig()` with the argument of `cfg` to initialize the new configuration.

When updating the replica configuration object, access the replica set members in the [members](#) (page 474) array with the **array index**. The array index begins with 0. Do **not** confuse this index value with the value of the [_id](#) (page 474) field in each document in the [members](#) (page 474) array.

If a member has [priority](#) (page 476) set to 0, it is ineligible to become *primary* and will not seek election. *Hidden members* (page 383), *delayed members* (page 383), and *arbiters* (page ??) all have [priority](#) (page 476) set to 0.

All members have a [priority](#) (page 476) equal to 1 by default.

The value of [priority](#) (page 476) can be any floating point (i.e. decimal) number between 0 and 1000. Priorities are only used to determine the preference in election. The priority value is used only in relation to other members. With the exception of members with a priority of 0, the absolute value of the [priority](#) (page 476) value is irrelevant.

Replica sets will preferentially elect and maintain the primary status of the member with the highest [priority](#) (page 476) setting.

Warning: Replica set reconfiguration can force the current primary to step down, leading to an election for primary in the replica set. Elections cause the current primary to close all open *client* connections. Perform routine replica set reconfiguration during scheduled maintenance windows.

See also:

The [Replica Reconfiguration Usage](#) (page 477) example revolves around changing the priorities of the [members](#) (page 474) of a replica set.

Prevent Secondary from Becoming Primary

To prevent a *secondary* member from ever becoming a *primary* in a *failover*, assign the secondary a priority of 0, as described here. You can set this “secondary-only mode” for any member of the *replica set*, except the current primary. For a detailed description of secondary-only members and their purposes, see [Priority 0 Replica Set Members](#) (page 382).

To configure a member as secondary-only, set its [priority](#) (page 476) value to 0 in the [members](#) (page 474) document in its replica set configuration. Any member with a [priority](#) (page 476) equal to 0 will never seek *election* (page 393) and cannot become primary in any situation.

```
{
  "_id" : <num>,
  "host" : <hostname:port>,
  "priority" : 0
}
```

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()`, and then you must *reconfigure the replica set* (page 477) with `rs.conf()` and `rs.reconfig()`.

Example

As an example of modifying member priorities, assume a four-member replica set. Use the following sequence of operations to modify member priorities in the `mongo` shell connected to the primary. Identify each member by its array index in the `members` (page 474) array:

```
cfg = rs.conf()
cfg.members[0].priority = 2
cfg.members[1].priority = 1
cfg.members[2].priority = 0.5
cfg.members[3].priority = 0
rs.reconfig(cfg)
```

The sequence of operations reconfigures the set with the following priority settings:

- Member at 0 has a priority of 2 so that it becomes primary under most circumstances.
- Member at 1 has a priority of 1, which is the default value. Member 1 becomes primary if no member with a *higher* priority is eligible.
- Member at 2 has a priority of 0.5, which makes it less likely to become primary than other members but doesn't prohibit the possibility.
- Member at 3 has a priority of 0. Member at 3 **cannot** become the *primary* member under any circumstances.

When updating the replica configuration object, access the replica set members in the `members` (page 474) array with the **array index**. The array index begins with 0. Do **not** confuse this index value with the value of the `_id` (page 474) field in each document in the `members` (page 474) array.

Warning:

- The `rs.reconfig()` shell method can force the current primary to step down, which causes an *election* (page 393). 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 427) to ensure that members can quickly obtain a majority of votes in an election for primary.

Related Documents

- [priority](#) (page 476)
- [Adjust Priority for Replica Set Member](#) (page 434)
- [Replica Set Reconfiguration](#) (page 477)
- [Replica Set Elections](#) (page 393)

Configure a Hidden Replica Set Member

Hidden members are part of a *replica set* but cannot become *primary* and are invisible to client applications. Hidden members do, however, vote in [elections](#) (page 393). For a detailed description of hidden members and their purposes, see [Hidden Replica Set Members](#) (page 383).

If the [chainingAllowed](#) (page 477) 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 452)

To configure a secondary member as hidden, set its [priority](#) (page 476) value to 0 and set its [hidden](#) (page 475) value to `true` in its member configuration:

```
{
  "_id" : <num>
  "host" : <hostname:port>,
  "priority" : 0,
  "hidden" : true
}
```

Example

The following example hides the secondary member currently at the index 0 in the [members](#) (page 474) 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](#) (page 474) 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](#) (page 474) array with the **array index**. The array index begins with 0. Do **not** confuse this index value with the value of the [_id](#) (page 474) field in each document in the [members](#) (page 474) array.

Warning:

- The `rs.reconfig()` shell method can force the current primary to step down, which causes an [election](#) (page 393). 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 427) to ensure that members can quickly obtain a majority of votes in an election for primary.

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.

Related Documents

- [Replica Set Reconfiguration](#) (page 477)
- [Replica Set Elections](#) (page 393)
- [Read Preference](#) (page 401)

Configure a Delayed Replica Set Member

To configure a delayed secondary member, set its [priority](#) (page 476) value to 0, its [hidden](#) (page 475) value to `true`, and its [slaveDelay](#) (page 476) value to the number of seconds to delay.

Important: The length of the secondary [slaveDelay](#) (page 476) must fit within the window of the oplog. If the oplog is shorter than the [slaveDelay](#) (page 476) 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 [Delayed Replica Set Members](#) (page 383).

Example

The following example sets a 1-hour delay on a secondary member currently at the index 0 in the [members](#) (page 474) 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](#) (page 476) 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](#) (page 474) array with the **array index**. The array index begins with 0. Do **not** confuse this index value with the value of the [_id](#) (page 474) field in each document in the [members](#) (page 474) array.

Warning:

- The `rs.reconfig()` shell method can force the current primary to step down, which causes an [election](#) (page 393). 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 427) to ensure that members can quickly obtain a majority of votes in an election for primary.

Related Documents

- [slaveDelay](#) (page 476)
- [Replica Set Reconfiguration](#) (page 477)
- [Oplog Size](#) (page 406)

- [Change the Size of the Oplog](#) (page 441) tutorial
- [Replica Set Elections](#) (page 393)

Configure Non-Voting Replica Set Member

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](#) (page 476) 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](#) (page 474) 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](#) (page 474) 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](#) (page 474) array with the **array index**. The array index begins with 0. Do **not** confuse this index value with the value of the [_id](#) (page 474) field in each document in the [members](#) (page 474) array.

Warning:

- The `rs.reconfig()` shell method can force the current primary to step down, which causes an [election](#) (page 393). 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 427) 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](#) (page 476) to control which members are more likely to become primary.

Related Documents

- [votes](#) (page 476)
- [Replica Set Reconfiguration](#) (page 477)
- [Replica Set Elections](#) (page 393)

Convert a Secondary to an Arbiter

- Convert Secondary to Arbiter and Reuse the Port Number (page 439)
- Convert Secondary to Arbiter Running on a New Port Number (page 440)

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](#) (page 393), you may convert the secondary to an *arbiter* (page ??) 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 439).

- 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 440).

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.

8.3.3 Replica Set Maintenance Tutorials

The following tutorials provide information in maintaining existing replica sets.

[Change the Size of the Oplog \(page 441\)](#) Increase the size of the *oplog* which logs operations. In most cases, the default oplog size is sufficient.

[Force a Member to Become Primary](#) (page 443) Force a replica set member to become primary.

[Resync a Member of a Replica Set](#) (page 445) 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 446) 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 450) Reconfigure a replica set when a majority of replica set members are down or unreachable.

[Manage Chained Replication](#) (page 452) 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 453) Update the replica set configuration to reflect changes in members' hostnames.

[Configure a Secondary's Sync Target](#) (page 457) Specify the member that a secondary member synchronizes from.

Change the Size of the Oplog

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](#) (page 406) 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 [Oplog Size](#) (page 406). For details how oplog size affects *delayed members* and affects *replication lag*, see [Delayed Replica Set Members](#) (page 383).

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 `--rep1Set` 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')
```

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.

Force a Member to Become Primary

Synopsis

You can force a *replica set* member to become *primary* by giving it a higher [priority](#) (page 476) value than any other member in the set.

Optionally, you also can force a member never to become primary by setting its [priority](#) (page 476) value to 0, which means the member can never seek [election](#) (page 393) as primary. For more information, see [Priority 0 Replica Set Members](#) (page 382).

Procedures

Force a Member to be Primary by Setting its Priority High Changed in version 2.0.

For more information on priorities, see [priority](#) (page 476).

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](#) (page 477).

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 the mongo shell, 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)
```

This sets m3.example.net to have a higher `local.system.replset.members[n].priority` (page 476) 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 an election in which m3.example.net becomes primary based on its `priority` (page 476) 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:1000000, force:1})
```

This prevents m1.example.net from being primary for 1,000,000 seconds, 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

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* (page 408).

This tutorial addressed 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 [Automatically Sync a Member](#) (page 445).
- 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 446).

Automatically Sync a Member

This procedure relies on MongoDB’s regular process for *initial sync* (page 408). This will store the current data on the member. For an overview of MongoDB initial sync process, see the [Replication Processes](#) (page 406) section.

To sync or resync a member:

1. If the member is an existing member, do the following:
 - (a) 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.
 - (b) 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.
2. Start the mongod instance on the member. For example:

```
mongod --dbpath /data/db/ --replSet rsProduction
```

At this point, the mongod will perform an initial sync. The length of the initial sync may 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 to for the data files, **only a snapshot backup**. For approaches to capture a consistent snapshot of a running `mongod` instance, see the [Backup Strategies for MongoDB Systems](#) (page 134) 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

- Differences Between Read Preferences and Write Concerns (page 446)
- Add Tag Sets to a Replica Set (page 447)
- Custom Multi-Datacenter Write Concerns (page 448)
- Configure Tag Sets for Functional Segregation of Read and Write Operations (page 449)

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` (page 476) sub-document.

This section introduces the configuration of tag sets. For an overview on tag sets and their use, see [Replica Set Write Concern](#) (page 47) and [Tag Sets](#) (page 403).

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` (page 477) 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.runCommand( { getLastError: 1, 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` (page 477) 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 concern operation will only return after the write operation propagates to at least two different racks in the each facility:

```
db.runCommand( { getLastError: 1, 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:¹¹

```
{ "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` (page 477) 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` (page 477) value to configure two write concern modes:

```
conf.settings = {
    "getLastErrorModes" : {
        "ssd" : {
            "ssd" : 1
        },
        "MultipleDC" : {
            "dc.va" : 1,
```

¹¹ Since read preferences and write concerns use the value of fields in tag sets differently, larger deployments may have some redundancy.

```
        "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 operation, to ensure that a write operation propagates to each data center.

```
db.runCommand( { getLastError: 1, 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

To reconfigure a *replica set* when a **minority** of members are unavailable, use the `rs.reconfig()` operation on the current *primary*, following the example in the [Replica Set Reconfiguration Procedure](#) (page 477).

This document provides the following options for re-configuring a replica set when a **majority** of members are *not* accessible:

- [Reconfigure by Forcing the Reconfiguration](#) (page 450)
- [Reconfigure by Replacing the Replica Set](#) (page 451)

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](#) (page 393) 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. 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` (page 474) 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]]
```

- 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.

- 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 450).

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 [Example Reconfiguration Operations](#) (page 477).

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.

- Stop the surviving `mongod` instances. To ensure a clean shutdown, use an existing *control 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()
```

- 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.

- Restart one of the `mongod` instances *without* the `--rep1Set` 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 *control 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 \(page 445\)](#)

Manage Chained Replication

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` (page 477) setting in [Replica Set Configuration \(page 474\)](#) 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 `replicaSetSyncFrom` 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` (page 477) field in *Replica Set Configuration* (page 474) to `false`.

You can use the following sequence of commands to set `chainingAllowed` (page 477) 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` sub-document. If they do, skip this step.

Warning: To avoid data loss, skip this step if the configuration settings contain the `settings` sub-document.

If the current configuration settings **do not** contain the `settings` sub-document, create the sub-document by issuing the following command:

```
cfg.settings = { }
```

3. Issue the following sequence of commands to set `chainingAllowed` (page 477) to `false`:

```
cfg.settings.chainingAllowed = false
rs.reconfig(cfg)
```

Re-enable Chained Replication

To re-enable chained replication, set `chainingAllowed` (page 477) 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

- [Overview \(page 454\)](#)
- [Assumptions \(page 454\)](#)
- [Change Hostnames while Maintaining Replica Set Availability \(page 455\)](#)
- [Change All Hostnames at the Same Time \(page 456\)](#)

For most *replica sets*, the hostnames in the `host` (page 475) 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` (page 475) 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` (page 475) field. Use either of the following approaches:

- *Change hostnames without disrupting availability* (page 455). 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 456). This approach has a shorter maintenance window, but the replica set will be unavailable during the operation.

See also:

[Replica Set Reconfiguration Process](#) (page 477), [Deploy a Replica Set](#) (page 416), and [Add Members to a Replica Set](#) (page 429).

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 454).

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](#) (page 474) 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 [Example Reconfiguration Operations](#) (page 477).

- (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 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](#) (page 474) 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" : [
    {
      "host" : "mongodb0.example.net:27017",
      "priority" : 1
    }
  ]
}
```

```
        "_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 454).

1. Stop all members in the *replica set*.
2. Restart each member *on a different port* and *without* using the `--rep1Set` 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 `--rep1Set` option. For example:

```
mongod --dbpath /data/db1/ --port 27017 --rep1Set 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

To override the default sync target selection logic, you may manually configure a *secondary* member's sync target for pulling *oplog* entries temporarily. The following operations provide access to this functionality:

- replSetSyncFrom command, or
- rs.syncFrom() helper in the mongo shell

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.

Note: 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.

8.3.4 Troubleshoot Replica Sets

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 <http://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 `db.printSlaveReplicationInfo()` method.

The returned document displays the `syncedTo` value for each member, which shows you when each member last read from the oplog, as shown in the following example:

```
source: m1.example.net:30001
syncedTo: Tue Oct 02 2012 11:33:40 GMT-0400 (EDT)
          = 7475 secs ago (2.08hrs)
source: m2.example.net:30002
syncedTo: Tue Oct 02 2012 11:33:40 GMT-0400 (EDT)
          = 7475 secs ago (2.08hrs)
```

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 Management Service](#)¹². For more information see the [documentation for MMS](#)¹³.

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 vitalized 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` (page 44) to require confirmation of replication to secondaries, as described in [replica](#)

¹²<http://mms.mongodb.com/>

¹³<http://mms.mongodb.com/help/>

set write concern (page 47). 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.

- **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* (page 45), the secondaries will not be able to read the oplog fast enough to keep up with changes.

To prevent this, require *write acknowledgment or journaled write concern* (page 44) 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* (page 47)
- *Replica Set Write Concern* (page 50)
- *Oplog Size* (page 406)

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 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 from 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`' (page 476) 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 only as long as two members can connect to each other. If two you reboot the two secondaries once, the primary steps down and becomes a secondary. Until the at least one secondary becomes available, the set has no primary and cannot elect a new primary.

For more information on votes, see *Replica Set Elections* (page 393). For related information on connection errors, see *Does TCP keepalive time affect sharded clusters and replica sets?* (page 607).

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 `db.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:

- *Oplog Size* (page 406),
- *Delayed Replica Set Members* (page 383), and
- *Check the Replication Lag* (page 458).

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 441) 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`

The *duplicate key on local.slaves* error, occurs when a *secondary* or *slave* changes its hostname and the *primary* or *master* tries to update its `local.slaves` collection with the new name. The update fails because it contains the same `_id` value as the document containing the previous hostname. The error itself will resemble the following.

```
exception 11000 E11000 duplicate key error index: local.slaves.$_id_ dup key: { : ObjectId('<object>')}
```

This is a benign error and does not affect replication operations on the *secondary* or *slave*.

To prevent the error from appearing, drop the `local.slaves` collection from the *primary* or *master*, with the following sequence of operations in the `mongo` shell:

```
use local
db.slaves.drop()
```

The next time a *secondary* or *slave* polls the *primary* or *master*, the *primary* or *master* recreates the `local.slaves` collection.

8.4 Replication Reference

8.4.1 Replication Methods in the mongo Shell

Name	Description
<code>rs.add()</code>	Adds a member to a replica set.
<code>rs.addArb()</code>	Adds an <i>arbiter</i> to a replica set.
<code>rs.conf()</code>	Returns the replica set configuration document.
<code>rs.freeze()</code>	Prevents the current member from seeking election as primary for a period of time.
<code>rs.help()</code>	Returns basic help text for <i>replica set</i> functions.
<code>rs.initiate()</code>	Initializes a new replica set.
<code>rs.reconfig()</code>	Re-configures a replica set by applying a new replica set configuration object.
<code>rs.remove()</code>	Remove a member from a replica set.
<code>rs.slaveOk()</code>	Sets the <code>slaveOk</code> property for the current connection. Deprecated. Use <code>readPref()</code> and <code>Mongo.setReadPref()</code> to set <i>read preference</i> .
<code>rs.status()</code>	Returns a document with information about the state of the replica set.
<code>rs.stepDown()</code>	Causes the current <i>primary</i> to become a <i>secondary</i> which forces an <i>election</i> .
<code>rs.syncFrom()</code>	Sets the member that this replica set member will sync from, overriding the default sync target selection logic.

8.4.2 Replication Database Commands

Name	Description
<code>replSetFreeze</code>	Prevents the current member from seeking election as <i>primary</i> for a period of time.
<code>replSetGetStatus</code>	Returns a document that reports on the status of the replica set.
<code>replSetInitiate</code>	Initializes a new replica set.
<code>replSetMaintenance</code>	Enables or disables a maintenance mode, which puts a <i>secondary</i> node in a <code>RECOVERING</code> state.
<code>replSetReconfig</code>	Applies a new configuration to an existing replica set.
<code>replSetStepDown</code>	Forces the current <i>primary</i> to <i>step down</i> and become a <i>secondary</i> , forcing an election.
<code>replSetSyncFrom</code>	Explicitly override the default logic for selecting a member to replicate from.
<code>resync</code>	Forces a <code>mongod</code> to re-synchronize from the <i>master</i> . For master-slave replication only.
<code>applyOps</code>	Internal command that applies <i>oplog</i> entries to the current data set.
<code>isMaster</code>	Displays information about this member's role in the replica set, including whether it is the <i>master</i> .
<code>getoptime</code>	Internal command to support replication, returns the <i>optime</i> .

8.4.3 Replica Set Reference Documentation

Replica Set Commands (page 463) A quick reference for all *commands* and mongo shell methods that support replication.

Replica Set Configuration (page 474) Complete documentation of the *replica set* configuration object returned by `rs.conf()`.

The local Database (page 479) Complete documentation of the content of the `local` database that `mongod` instances use to support replication.

Replica Set Member States (page 481) Reference for the replica set member states.

Read Preference Reference (page 483) Complete documentation of the five read preference modes that the MongoDB drivers support.

Replica Set Commands

This reference collects documentation for all *JavaScript methods* (page 463) for the mongo shell that support *replica set* functionality, as well as all *database commands* (page 467) related to replication function.

See [Replication](#) (page 373), for a list of all replica set documentation.

JavaScript Methods

The following methods apply to replica sets. For a complete list of all methods, see <http://docs.mongodb.org/manual/reference/method/>.

`rs.status()`

Returns A *document* with status information.

This output reflects the current status of the replica set, using data derived from the heartbeat packets sent by the other members of the replica set.

This method provides a wrapper around the `replSetGetStatus` *database command*.

`db.isMaster()`

Returns A document that describes the role of the `mongod` instance.

If the `mongod` is a member of a *replica set*, then the `ismaster` and `secondary` fields report if the instance is the *primary* or if it is a *secondary* member of the replica set.

See

`isMaster` for the complete documentation of the output of `db.isMaster()`.

Description

`rs.initiate(configuration)`

Initiates a *replica set*. Optionally takes a configuration argument in the form of a *document* that holds the configuration of a replica set.

The `rs.initiate()` method has the following parameter:

param document configuration A *document* that specifies *configuration settings* (page 474) for the new replica set. If a configuration is not specified, MongoDB uses a default configuration.

The `rs.initiate()` method provides a wrapper around the “`replSetInitiate`” *database command*.

Replica Set Configuration

See [Member Configuration Tutorials](#) (page 433) and [Replica Set Configuration](#) (page 474) for examples of replica set configuration and invitation objects.

`rs.conf()`

Returns a *document* that contains the current *replica set* configuration document.

See [Replica Set Configuration](#) (page 474) for more information on the replica set configuration document.

`rs.config()`

`rs.config()` is an alias of `rs.conf()`.

Definition

`rs.reconfig(configuration, force)`

Initializes a new *replica set* configuration. Disconnects the shell briefly and forces a reconnection as the replica set renegotiates which member will be *primary*. As a result, the shell will display an error even if this command succeeds.

param document configuration A *document* that specifies the configuration of a replica set.

param document force “If set as { `force: true` }, this forces the replica set to accept the new configuration even if a majority of the members are not accessible. Use with caution, as this can lead to term:*rollback* situations.”

`rs.reconfig()` overwrites the existing replica set configuration. Retrieve the current configuration object with `rs.conf()`, modify the configuration as needed and then use `rs.reconfig()` to submit the modified configuration object.

`rs.reconfig()` provides a wrapper around the “`replSetReconfig`” *database command*.

Examples

To reconfigure a replica set, use the following sequence of operations:

```
conf = rs.conf()  
  
// modify conf to change configuration  
  
rs.reconfig(conf)
```

If you want to force the reconfiguration if a majority of the set is not connected to the current member, or you are issuing the command against a secondary, use the following form:

```
conf = rs.conf()  
  
// modify conf to change configuration  
  
rs.reconfig(conf, { force: true } )
```

Warning: Forcing a `rs.reconfig()` can lead to *rollback* situations and other difficult to recover from situations. Exercise caution when using this option.

See also:

[Replica Set Configuration](#) (page 474) and [Replica Set Tutorials](#) (page 415).

Definition

```
rs.add(host, arbiterOnly)
```

Adds a member to a *replica set*.

param string,document host The new member to add to the replica set. If a string, specifies the hostname and optionally the port number for the new member. If a document, specifies a replica set members document, as found in the `members` (page 474) array. To view a replica set's members array, run `rs.conf()`.

param boolean arbiterOnly Applies only if the `<host>` value is a string. If `true`, the added host is an arbiter.”

You may specify new hosts in one of two ways:

- 1.as a “hostname” with an optional port number to use the default configuration as in the [Add a Member to an Existing Replica Set](#) (page 430) example.

- 2.as a configuration *document*, as in the [Configure and Add a Member](#) (page 431) example.

This function will disconnect the shell briefly and forces a reconnection as the replica set renegotiates which member will be *primary*. As a result, the shell will display an error even if this command succeeds.

`rs.add()` provides a wrapper around some of the functionality of the “`replSetReconfig`” *database command* and the corresponding shell helper `rs.reconfig()`. See the [Replica Set Configuration](#) (page 474) document for full documentation of all replica set configuration options.

Example

To add a `mongod` accessible on the default port 27017 running on the host `mongodb3.example.net`, use the following `rs.add()` invocation:

```
rs.add('mongodb3.example.net:27017')
```

If `mongodb3.example.net` is an arbiter, use the following form:

```
rs.add('mongodb3.example.net:27017', true)
```

To add `mongodb3.example.net` as a *secondary-only* (page 382) member of set, use the following form of `rs.add()`:

```
rs.add( { "_id": 3, "host": "mongodb3.example.net:27017", "priority": 0 } )
```

Replace, 3 with the next unused `_id` value in the replica set. See `rs.conf()` to see the existing `_id` values in the replica set configuration document.

See the [Replica Set Configuration](#) (page 474) and [Replica Set Tutorials](#) (page 415) documents for more information.

Description

```
rs.addArb(host)
```

Adds a new *arbiter* to an existing replica set.

The `rs.addArb()` method takes the following parameter:

param string host Specifies the hostname and optionally the port number of the arbiter member to add to replica set.

This function briefly disconnects the shell and forces a reconnection as the replica set renegotiates which member will be *primary*. As a result, the shell displays an error even if this command succeeds.

Description

`rs.stepDown(seconds)`

Forces the current *replica set* member to step down as *primary* and then attempt to avoid election as primary for the designated number of seconds. Produces an error if the current member is not the primary.

The `rs.stepDown()` method has the following parameter:

param number seconds The duration of time that the stepped-down member attempts to avoid re-election as primary. If this parameter is not specified, the method uses the default value of 60 seconds.

This function disconnects the shell briefly and forces a reconnection as the replica set renegotiates which member will be primary. As a result, the shell will display an error even if this command succeeds.

`rs.stepDown()` provides a wrapper around the *database command* `replSetStepDown`.

Description

`rs.freeze(seconds)`

Makes the current *replica set* member ineligible to become *primary* for the period specified.

The `rs.freeze()` method has the following parameter:

param number seconds The duration the member is ineligible to become primary.

`rs.freeze()` provides a wrapper around the *database command* `replSetFreeze`.

Definition

`rs.remove(hostname)`

Removes the member described by the `hostname` parameter from the current *replica set*. This function will disconnect the shell briefly and forces a reconnection as the *replica set* renegotiates which member will be *primary*. As a result, the shell will display an error even if this command succeeds.

The `rs.remove()` method has the following parameter:

param string hostname The hostname of a system in the replica set.

Note: Before running the `rs.remove()` operation, you must *shut down* the replica set member that you're removing.

Changed in version 2.2: This procedure is no longer required when using `rs.remove()`, but it remains good practice.

`rs.slaveOk()`

Provides a shorthand for the following operation:

```
db.getMongo().setSlaveOk()
```

This allows the current connection to allow read operations to run on *secondary* members. See the `readPref()` method for more fine-grained control over *read preference* (page 401) in the mongo shell.

`db.isMaster()`

Returns A document that describes the role of the mongod instance.

If the mongod is a member of a *replica set*, then the `ismaster` and `secondary` fields report if the instance is the *primary* or if it is a *secondary* member of the replica set.

See

`isMaster` for the complete documentation of the output of `db.isMaster()`.

`rs.help()`

Returns a basic help text for all of the *replication* (page 373) related shell functions.

`rs.syncFrom()`

New in version 2.2.

Provides a wrapper around the `replSetSyncFrom`, which allows administrators to configure the member of a replica set that the current member will pull data from. Specify the name of the member you want to replicate from in the form of [hostname] : [port].

See `replSetSyncFrom` for more details.

Database Commands

The following commands apply to replica sets. For a complete list of all commands, see <http://docs.mongodb.org/manual/reference/command>.

Definition

isMaster

`isMaster` returns a document that describes the role of the mongod instance.

If the instance is a member of a replica set, then `isMaster` returns a subset of the replica set configuration and status including whether or not the instance is the *primary* of the replica set.

When sent to a mongod instance that is not a member of a replica set, `isMaster` returns a subset of this information.

MongoDB *drivers* and *clients* use `isMaster` to determine the state of the replica set members and to discover additional members of a *replica set*.

The `db.isMaster()` method in the mongo shell provides a wrapper around `isMaster`.

The command takes the following form:

```
{ isMaster: 1 }
```

See also:

`db.isMaster()`

Output

All Instances The following `isMaster` fields are common across all roles:

isMaster.ismaster

A boolean value that reports when this node is writable. If `true`, then this instance is a *primary* in a *replica set*, or a *master* in a master-slave configuration, or a `mongos` instance, or a standalone `mongod`.

This field will be `false` if the instance is a *secondary* member of a replica set or if the member is an *arbiter* of a replica set.

isMaster.maxBsonObjectSize

The maximum permitted size of a *BSON object* in bytes for this `mongod` process. If not provided, clients should assume a max size of “`4 * 1024 * 1024`”.

isMaster.maxMessageSizeBytes

New in version 2.4.

The maximum permitted size of a *BSON wire protocol message*. The default value is `48000000` bytes.

isMaster.localTime

New in version 2.2.

Returns the local server time in UTC. This value is an *ISO date*.

Sharded Instances `mongos` instances add the following field to the `isMaster` response document:

isMaster.msg

Contains the value `isdbgrid` when `isMaster` returns from a `mongos` instance.

Replica Sets `isMaster` contains these fields when returned by a member of a replica set:

isMaster.setName

The name of the current :replica set.

isMaster.secondary

A boolean value that, when `true`, indicates if the `mongod` is a *secondary* member of a *replica set*.

isMaster.hosts

An array of strings in the format of "`[hostname] : [port]`" that lists all members of the *replica set* that are neither *hidden*, *passive*, nor *arbiters*.

Drivers use this array and the `isMaster.passives` to determine which members to read from.

isMaster.passives

An array of strings in the format of "`[hostname] : [port]`" listing all members of the *replica set* which have a `priorities` (page 476) of 0.

This field only appears if there is at least one member with a `priorities` (page 476) of 0.

Drivers use this array and the `isMaster.hosts` to determine which members to read from.

isMaster.arbiters

An array of strings in the format of "`[hostname] : [port]`" listing all members of the *replica set* that are *arbiters*.

This field only appears if there is at least one arbiter in the replica set.

isMaster.primary

A string in the format of "`[hostname] : [port]`" listing the current *primary* member of the replica set.

isMaster.arbiterOnly

A boolean value that, when `true`, indicates that the current instance is an *arbiter*. The `arbiterOnly` field is only present, if the instance is an arbiter.

isMaster.passive

A boolean value that, when `true`, indicates that the current instance is *hidden*. The `passive` field is only present for hidden members.

isMaster.hidden

A boolean value that, when `true`, indicates that the current instance is *hidden*. The `hidden` field is only present for hidden members.

isMaster.tags

A document that lists any tags assigned to this member. This field is only present if there are tags assigned to the member. See [Configure Replica Set Tag Sets](#) (page 446) for more information.

isMaster.me

The `[hostname] : [port]` of the member that returned `isMaster`.

resync

The `resync` command forces an out-of-date slave mongod instance to re-synchronize itself. Note that this command is relevant to master-slave replication only. It does not apply to replica sets.

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

replSetFreeze

The `replSetFreeze` command prevents a replica set member from seeking election for the specified number of seconds. Use this command in conjunction with the `replSetStepDown` command to make a different node in the replica set a primary.

The `replSetFreeze` command uses the following syntax:

```
{ replSetFreeze: <seconds> }
```

If you want to unfreeze a replica set member before the specified number of seconds has elapsed, you can issue the command with a `seconds` value of `0`:

```
{ replSetFreeze: 0 }
```

Restarting the mongod process also unfreezes a replica set member.

`replSetFreeze` is an administrative command, and you must issue it against the *admin database*.

Definition

replSetGetStatus

The `replSetGetStatus` command returns the status of the replica set from the point of view of the current server. You must run the command against the *admin database*. The command has the following prototype format:

```
{ replSetGetStatus: 1 }
```

The value specified does not affect the output of the command. Data provided by this command derives from data included in heartbeats sent to the current instance by other members of the replica set. Because of the frequency of heartbeats, these data can be several seconds out of date.

You can also access this functionality through the `rs.status()` helper in the mongo shell.

The mongod must have replication enabled and be a member of a replica set for the `replSetGetStatus` to return successfully.

Output

replSetGetStatus.set

The set value is the name of the replica set, configured in the `rep1Set` setting. This is the same value as `_id` (page 474) in `rs.conf()`.

replSetGetStatus.date

The value of the date field is an *ISODate* of the current time, according to the current server. Compare this to the value of the `lastHeartbeat` to find the operational lag between the current host and the other hosts in the set.

replSetGetStatus.myState

The value of `myState` is an integer between 0 and 10 that represents the *replica state* (page 481) of the current member.

replSetGetStatus.members

The members field holds an array that contains a document for every member in the replica set.

replSetGetStatus.members.name

The name field holds the name of the server.

replSetGetStatus.members.self

The self field is only included in the document for the current `mongod` instance in the members array. Its value is true.

replSetGetStatus.members.errmsg

This field contains the most recent error or status message received from the member. This field may be empty (e.g. "") in some cases.

replSetGetStatus.members.health

The health value is only present for the other members of the replica set (i.e. not the member that returns `rs.status()`). This field conveys if the member is up (i.e. 1) or down (i.e. 0).

replSetGetStatus.members.state

The value of state is an array of documents, each containing an integer between 0 and 10 that represents the *replica state* (page 481) of the corresponding member.

replSetGetStatus.members.stateStr

A string that describes state.

replSetGetStatus.members.uptime

The uptime field holds a value that reflects the number of seconds that this member has been online.

This value does not appear for the member that returns the `rs.status()` data.

replSetGetStatus.members.optime

A document that contains information regarding the last operation from the operation log that this member has applied.

replSetGetStatus.members.optime.t

A 32-bit timestamp of the last operation applied to this member of the replica set from the *oplog*.

replSetGetStatus.members.optime.i

An incremented field, which reflects the number of operations in since the last time stamp. This value only increases if there is more than one operation per second.

replSetGetStatus.members.optimeDate

An *ISODate* formatted date string that reflects the last entry from the *oplog* that this member applied. If this differs significantly from `lastHeartbeat` this member is either experiencing “replication lag” or there have not been any new operations since the last update. Compare `members.optimeDate` between all of the members of the set.

replSetGetStatus.members.lastHeartbeat

The `lastHeartbeat` value provides an `ISODate` formatted date of the last heartbeat received from this member. Compare this value to the value of the `date` field to track latency between these members.

This value does not appear for the member that returns the `rs.status()` data.

replSetGetStatus.members.pingMS

The `pingMS` represents the number of milliseconds (ms) that a round-trip packet takes to travel between the remote member and the local instance.

This value does not appear for the member that returns the `rs.status()` data.

replSetGetStatus.syncingTo

The `syncingTo` field is only present on the output of `rs.status()` on `secondary` and recovering members, and holds the hostname of the member from which this instance is syncing.

replicaSetInitiate

The `replicaSetInitiate` command initializes a new replica set. Use the following syntax:

```
{ replicaSetInitiate : <config_document> }
```

The `<config_document>` is a *document* that specifies the replica set's configuration. For instance, here's a config document for creating a simple 3-member replica set:

```
{
  _id : <setname>,
  members : [
    {_id : 0, host : <host0>},
    {_id : 1, host : <host1>},
    {_id : 2, host : <host2>},
  ]
}
```

A typical way of running this command is to assign the config document to a variable and then to pass the document to the `rs.initiate()` helper:

```
config = {
  _id : "my_replica_set",
  members : [
    {_id : 0, host : "rs1.example.net:27017"},
    {_id : 1, host : "rs2.example.net:27017"},
    {_id : 2, host : "rs3.example.net", arbiterOnly: true},
  ]
}

rs.initiate(config)
```

Notice that omitting the port cause the host to use the default port of 27017. Notice also that you can specify other options in the config documents such as the `arbiterOnly` setting in this example.

See also:

[Replica Set Configuration](#) (page 474), [Replica Set Tutorials](#) (page 415), and [Replica Set Reconfiguration](#) (page 477).

replicaSetMaintenance

The `replicaSetMaintenance` admin command enables or disables the maintenance mode for a `secondary` member of a *replica set*.

The command has the following prototype form:

```
{ replSetMaintenance: <boolean> }
```

Consider the following behavior when running the `replSetMaintenance` command:

- You cannot run the command on the Primary.
- You must run the command against the `admin` database.
- When enabled `replSetMaintenance: true`, the member enters the `RECOVERING` state. While the secondary is `RECOVERING`:
 - The member is not accessible for read operations.
 - The member continues to sync its *oplog* from the Primary.

Important: On secondaries, the `compact` command forces the secondary to enter `RECOVERING` (page 482) state. This prevents clients from reading during compaction. Once the operation finishes, the secondary returns to `SECONDARY` (page 481) state.

See [Replica Set Member States](#) (page 481) for more information about replica set member states. Refer to the “partial script for automating step down and compaction¹⁴” for an example of this procedure.

`replSetReconfig`

The `replSetReconfig` command modifies the configuration of an existing replica set. You can use this command to add and remove members, and to alter the options set on existing members. Use the following syntax:

```
{ replSetReconfig: <new_config_document>, force: false }
```

You may also run the command using the shell’s `rs.reconfig()` method.

Be aware of the following `replSetReconfig` behaviors:

- You must issue this command against the *admin database* of the current primary member of the replica set.
- You can optionally force the replica set to accept the new configuration by specifying `force: true`. Use this option if the current member is not primary or if a majority of the members of the set are not accessible.

Warning: Forcing the `replSetReconfig` command can lead to a *rollback* situation. Use with caution.

Use the `force` option to restore a replica set to new servers with different hostnames. This works even if the set members already have a copy of the data.

- A majority of the set’s members must be operational for the changes to propagate properly.
- This command can cause downtime as the set renegotiates primary-status. Typically this is 10-20 seconds, but could be as long as a minute or more. Therefore, you should attempt to reconfigure only during scheduled maintenance periods.
- In some cases, `replSetReconfig` forces the current primary to step down, initiating an election for primary among the members of the replica set. When this happens, the set will drop all current connections.

Note: `replSetReconfig` obtains a special mutually exclusive lock to prevent more than one `replSetReconfig` operation from occurring at the same time.

¹⁴<https://github.com/mongodb/mongo-snippets/blob/master/js/compact-example.js>

Description

`replSetSyncFrom`

New in version 2.2.

Explicitly configures which host the current mongod pulls *oplog* entries from. This operation is useful for testing different patterns and in situations where a set member is not replicating from the desired host.

The `replSetSyncFrom` command has the following form:

```
{ replSetSyncFrom: "hostname<:port>" }
```

The `replSetSyncFrom` command has the following field:

field string `replSetSyncFrom` The name and port number of the replica set member that this member should replicate from. Use the `[hostname] : [port]` form.

The Target Member

The member to replicate from must be a valid source for data in the set. The member cannot be:

- The same as the mongod on which you run `replSetSyncFrom`. In other words, a member cannot replicate from itself.
- An arbiter, because arbiters do not hold data.
- A member that does not build indexes.
- An unreachable member.
- A mongod instance that is not a member of the same replica set.

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.

Run from the mongo Shell

To run the command in the mongo shell, use the following invocation:

```
db.adminCommand( { replSetSyncFrom: "hostname<:port>" } )
```

You may also use the `rs.syncFrom()` helper in the mongo shell in an operation with the following form:

```
rs.syncFrom("hostname<:port>")
```

Note: `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.

Replica Set Configuration

Synopsis

This reference provides an overview of replica set configuration options and settings.

Use `rs.conf()` in the mongo shell to retrieve this configuration. Note that default values are not explicitly displayed.

Example Configuration Document

The following document provides a representation of a replica set configuration document. Angle brackets (e.g. < and >) enclose all optional fields.

```
{
  _id : <setname>,
  version: <int>,
  members: [
    {
      _id : <ordinal>,
      host : hostname<:port>,
      <arbiterOnly : <boolean>>,
      <buildIndexes : <boolean>>,
      <hidden : <boolean>>,
      <priority: <priority>>,
      <tags: { <document> }>,
      <slaveDelay : <number>>,
      <votes : <number>>
    }
  ,
  ...
  ],
  <settings: {
    <getLastDefaults : <lasterrdefaults>>,
    <chainingAllowed : <boolean>>
    <getLastModes : <modes>>
  }>
}
```

Configuration Variables

`local.system.replset._id`

Type: string

Value: <setname>

An `_id` field holding the name of the replica set. This reflects the set name configured with `rep1Set` or `mongod --rep1Set`.

`local.system.replset.members`

Type: array

Contains an array holding an embedded *document* for each member of the replica set. The `members` document contains a number of fields that describe the configuration of each member of the replica set.

The `members` (page 474) field in the replica set configuration document is a zero-indexed array.

`local.system.replset.members[n]._id`

Type: ordinal

Provides the zero-indexed identifier of every member in the replica set.

Note: When updating the replica configuration object, access the replica set members in the `members` (page 474) array with the **array index**. The array index begins with 0. Do **not** confuse this index value with the value of the `_id` (page 474) field in each document in the `members` (page 474) array.

`local.system.replset.members[n].host`

Type: <hostname>:<port>

Identifies the host name of the set member with a hostname and port number. This name must be resolvable for every host in the replica set.

Warning: `host` (page 475) cannot hold a value that resolves to `localhost` or the local interface unless *all* members of the set are on hosts that resolve to `localhost`.

`local.system.replset.members[n].arbiterOnly`

Optional.

Type: boolean

Default: false

Identifies an arbiter. For arbiters, this value is `true`, and is automatically configured by `rs.addArb()`.

`local.system.replset.members[n].buildIndexes`

Optional.

Type: boolean

Default: true

Determines whether the `mongod` builds *indexes* on this member. Do not set to `false` for instances that receive queries from clients.

Omitting index creation, and thus this setting, may be useful, **if**:

- You are only using this instance to perform backups using `mongodump`,
- this instance will receive no queries, *and*
- index creation and maintenance overburdens the host system.

If set to `false`, secondaries configured with this option *do* build indexes on the `_id` field, to facilitate operations required for replication.

Warning: You may only set this value when adding a member to a replica set. You may not reconfigure a replica set to change the value of the `buildIndexes` (page 475) field after adding the member to the set. `buildIndexes` (page 475) is only valid when `priority` is 0 to prevent these members from becoming `primary`. Make all instances that do not build indexes hidden.

Other secondaries cannot replicate from a members where `buildIndexes` (page 475) is false.

`local.system.replset.members[n].hidden`

Optional.

Type: boolean

Default: false

When this value is `true`, the replica set hides this instance, and does not include the member in the output of `db.isMaster()` or `isMaster`. This prevents read operations (i.e. queries) from ever reaching this host by way of secondary *read preference*.

See also:

[Hidden Replica Set Members](#) (page 383)

`local.system.replset.members[n].priority`

Optional.

Type: Number, between 0 and 100.0 including decimals.

Default: 1

Specify higher values to make a member *more* eligible to become *primary*, and lower values to make the member *less* eligible to become primary. Priorities are only used in comparison to each other. Members of the set will veto election requests from members when another eligible member has a higher priority value. Changing the balance of priority in a replica set will trigger an election.

A `priority` (page 476) of 0 makes it impossible for a member to become primary.

See also:

[priority](#) (page 476) and [Replica Set Elections](#) (page 393).

`local.system.replset.members[n].tags`

Optional.

Type: MongoDB Document

Default: none

Used to represent arbitrary values for describing or tagging members for the purposes of extending *write concern* to allow configurable data center awareness.

Use in conjunction with [getLastErrorModes](#) (page 477) and [getLastErrorDefaults](#) (page 477) and `db.getLastError()` (i.e. `getLastError`.)

For procedures on configuring tag sets, see [Configure Replica Set Tag Sets](#) (page 446).

Important: In tag sets, all tag values must be strings.

`local.system.replset.members[n].slaveDelay`

Optional.

Type: Integer. (seconds.)

Default: 0

Describes the number of seconds “behind” the primary that this replica set member should “lag.” Use this option to create [delayed members](#) (page 383), that maintain a copy of the data that reflects the state of the data set at some amount of time in the past, specified in seconds. Typically such delayed members help protect against human error, and provide some measure of insurance against the unforeseen consequences of changes and updates.

`local.system.replset.members[n].votes`

Optional.

Type: Integer

Default: 1

Controls the number of votes a server will cast in a [replica set election](#) (page 393). The number of votes each member has can be any non-negative integer, but it is highly recommended each member has 1 or 0 votes.

If you need more than 7 members in one replica set, use this setting to add additional non-voting members with a `votes` (page 476) value of 0.

For most deployments and most members, use the default value, 1, for `votes` (page 476).

`local.system.replset.settings`

Optional.

Type: MongoDB Document

The `settings` document configures options that apply to the whole replica set.

`local.system.replset.settings.chainingAllowed`

Optional.

Type: boolean

Default: true

New in version 2.2.4.

When `chainingAllowed` (page 477) is `true`, the replica set allows *secondary* members to replicate from other secondary members. When `chainingAllowed` (page 477) is `false`, secondaries can replicate only from the *primary*.

When you run `rs.config()` to view a replica set's configuration, the `chainingAllowed` (page 477) field appears only when set to `false`. If not set, `chainingAllowed` (page 477) is `true`.

See also:

[Manage Chained Replication](#) (page 452)

`local.system.replset.settings.getLastErrorDefaults`

Optional.

Type: MongoDB Document

Specify arguments to the `getLastError` that members of this replica set will use when no arguments to `getLastError` has no arguments. If you specify *any* arguments, `getLastError`, ignores these defaults.

`local.system.replset.settings.getLastErrorModes`

Optional.

Type: MongoDB Document

Defines the names and combination of `members` (page 474) for use by the application layer to guarantee *write concern* to database using the `getLastError` command to provide *data-center awareness*.

Example Reconfiguration Operations

Most modifications of *replica set* configuration use the `mongo` shell. Consider the following reconfiguration operation:

Example

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"
        }
    ]
}
```

The following reconfiguration operation updates the [priority](#) (page 476) of the replica set members:

```
cfg = rs.conf()
cfg.members[0].priority = 0.5
cfg.members[1].priority = 2
cfg.members[2].priority = 2
rs.reconfig(cfg)
```

First, this operation sets the local variable `cfg` to the current replica set configuration using the `rs.conf()` method. Then it adds priority values to the `cfg` *document* for the three sub-documents in the `members` (page 474) array, accessing each replica set member with the array index and **not** the replica set member's `_id` (page 474) field. Finally, it calls the `rs.reconfig()` method with the argument of `cfg` to initialize this new configuration. The replica set configuration after this operation will resemble the following:

```
{
    "_id" : "rs0",
    "version" : 1,
    "members" : [
        {
            "_id" : 0,
            "host" : "mongodb0.example.net:27017",
            "priority" : 0.5
        },
        {
            "_id" : 1,
            "host" : "mongodb1.example.net:27017",
            "priority" : 2
        },
        {
            "_id" : 2,
            "host" : "mongodb2.example.net:27017",
            "priority" : 1
        }
    ]
}
```

Using the “dot notation” demonstrated in the above example, you can modify any existing setting or specify any of optional [replica set configuration variables](#) (page 474). Until you run `rs.reconfig(cfg)` at the shell, no changes will take effect. You can issue `cfg = rs.conf()` at any time before using `rs.reconfig()` to undo your changes and start from the current configuration. If you issue `cfg` as an operation at any point, the mongo shell at any point will output the complete *document* with modifications for your review.

The `rs.reconfig()` operation has a “force” option, to make it possible to reconfigure a replica set if a majority of the replica set is not visible, and there is no *primary* member of the set. use the following form:

```
rs.reconfig(cfg, { force: true } )
```

Warning: Forcing a `rs.reconfig()` can lead to *rollback* situations and other difficult to recover from situations. Exercise caution when using this option.

Note: The `rs.reconfig()` shell method can force the current primary to step down and triggers an election in some situations. When the primary steps down, all clients will disconnect. This is by design. Since this typically takes 10-20 seconds, attempt to make such changes during scheduled maintenance periods.

The local Database

Overview

Every `mongod` instance has its own `local` database, which stores data used in the replication process, and other instance-specific data. The `local` database is invisible to replication: collections in the `local` database are not replicated.

In replication, the `local` database store stores internal replication data for each member of a *replica set*. The `local` stores the following collections:

Changed in version 2.4: When running with authentication (i.e. `auth`), authenticating to the `local` database is **not** equivalent to authenticating to the `admin` database. In previous versions, authenticating to the `local` database provided access to all databases.

Collection on all `mongod` Instances

`local.startup_log`

On startup, each `mongod` instance inserts a document into `startup_log` (page 479) with diagnostic information about the `mongod` instance itself and host information. `startup_log` (page 479) is a capped collection. This information is primarily useful for diagnostic purposes.

Example

Consider the following prototype of a document from the `startup_log` (page 479) collection:

```
{
    "_id" : "<string>",
    "hostname" : "<string>",
    "startTime" : ISODate("<date>"),
    "startTimeLocal" : "<string>",
    "cmdLine" : {
        "dbpath" : "<path>",
        "<option>" : <value>
    },
    "pid" : <number>,
    "buildinfo" : {
        "version" : "<string>",
        "gitVersion" : "<string>",
        "sysInfo" : "<string>",
        "loaderFlags" : "<string>",
        "compilerFlags" : "<string>",
        "allocator" : "<string>",
        "versionArray" : [ <num>, <num>, <...> ],
        "platform" : "<string>"
    }
}
```

```
        "javascriptEngine" : "<string>",
        "bits" : <number>,
        "debug" : <boolean>,
        "maxBsonObjectSize" : <number>
    }
}
```

Documents in the [startup_log](#) (page 479) collection contain the following fields:

local.startup_log._id

Includes the system hostname and a millisecond epoch value.

local.startup_log.hostname

The system's hostname.

local.startup_log.startTime

A UTC *ISODate* value that reflects when the server started.

local.startup_log.startTimeLocal

A string that reports the [startTime](#) (page 480) in the system's local time zone.

local.startup_log.cmdLine

A sub-document that reports the mongod runtime options and their values.

local.startup_log.pid

The process identifier for this process.

local.startup_log.buildinfo

A sub-document that reports information about the build environment and settings used to compile this mongod. This is the same output as `buildInfo`. See [buildInfo](#).

Collections on Replica Set Members

local.system.replset

[local.system.replset](#) (page 480) holds the replica set's configuration object as its single document. To view the object's configuration information, issue `rs.conf()` from the mongo shell. You can also query this collection directly.

local.oplog.rs

[local.oplog.rs](#) (page 480) is the capped collection that holds the *oplog*. You set its size at creation using the `oplogSize` setting. To resize the oplog after replica set initiation, use the [Change the Size of the Oplog](#) (page 441) procedure. For additional information, see the [Oplog Size](#) (page 406) section.

local.replset.invalid

This contains an object used internally by replica sets to track replication status.

local.slaves

This contains information about each member of the set and the latest point in time that this member has synced to. If this collection becomes out of date, you can refresh it by dropping the collection and allowing MongoDB to automatically refresh it during normal replication:

```
db.getSiblingDB("local").slaves.drop()
```

Collections used in Master/Slave Replication

In *master/slave* replication, the `local` database contains the following collections:

- On the master:

`local.oplog.$main`

This is the oplog for the master-slave configuration.

`local.slaves`

This contains information about each slave.

- On each slave:

`local.sources`

This contains information about the slave's master server.

Replica Set Member States

Members of replica sets have states that reflect the startup process, basic operations, and potential error states.

Number	Name	State Description
0	STARTUP (page 482)	Cannot vote. All members start up in this state. The mongod parses the replica set configuration document (page 433) while in STARTUP (page 482).
1	PRIMARY (page 481)	Can vote. The primary (page 378) is the only member to accept write operations.
2	SECONDARY (page 481)	Can vote. The secondary (page 378) replicates the data store.
3	RECOVERING (page 482)	Can vote. Members either perform startup self-checks, or transition from completing a rollback (page 397) or resync (page 445).
4	FATAL (page 482)	Cannot vote. Has encountered an unrecoverable error.
5	STARTUP2 (page 482)	Cannot vote. Forks replication and election threads before becoming a secondary.
6	UNKNOWN (page 482)	Cannot vote. Has never connected to the replica set.
7	ARBITER (page 481)	Can vote. Arbiters (page ??) do not replicate data and exist solely to participate in elections.
8	DOWN (page 482)	Cannot vote. Is not accessible to the set.
9	ROLLBACK (page 482)	Can vote. Performs a rollback (page 397).
10	SHUNNED (page 482)	Cannot vote. Was once in the replica set but has now been removed.

States

Core States

[PRIMARY](#)

Members in [PRIMARY](#) (page 481) state accept write operations. A replica set has only one primary at a time. A [SECONDARY](#) (page 481) member becomes primary after an [election](#) (page 393). Members in the [PRIMARY](#) (page 481) state are eligible to vote.

[SECONDARY](#)

Members in [SECONDARY](#) (page 481) state replicate the primary's data set and can be configured to accept read operations. Secondaries are eligible to vote in elections, and may be elected to the [PRIMARY](#) (page 481) state if the primary becomes unavailable.

ARBITER

Members in [ARBITER](#) (page 481) state do not replicate data or accept write operations. They are eligible to vote, and exist solely to break a tie during elections. Replica sets should only have a member in the [ARBITER](#) (page 481) state if the set would otherwise have an even number of members, and could suffer from tied elections. Like primaries, there should only be at most one arbiter in any replica set.

See [Replica Set Members](#) (page 378) for more information on core states.

Initialization States

STARTUP

Each member of a replica set starts up in [STARTUP](#) (page 482) state. mongod then loads that member's replica set configuration, and transitions the member's state to [STARTUP2](#) (page 482). Members in [STARTUP](#) (page 482) are not eligible to vote.

STARTUP2

Each member of a replica set enters the [STARTUP2](#) (page 482) state as soon as mongod finishes loading that member's configuration. While in the [STARTUP2](#) (page 482) state, the member creates threads to handle internal replication operations. Members are in the [STARTUP2](#) (page 482) state for a short period of time before entering the [RECOVERING](#) (page 482) state. Members in the [STARTUP2](#) (page 482) state are not eligible to vote.

RECOVERING

A member of a replica set enters [RECOVERING](#) (page 482) state when it is not ready to accept reads. The [RECOVERING](#) (page 482) state can occur during normal operation, and doesn't necessarily reflect an error condition. Members in the [RECOVERING](#) (page 482) state are eligible to vote in elections, but is not eligible to enter the [PRIMARY](#) (page 481) state.

During startup, members transition through [RECOVERING](#) (page 482) after [STARTUP2](#) (page 482) and before becoming [SECONDARY](#) (page 481).

During normal operation, if a *secondary* falls behind the other members of the replica set, it may need to [resync](#) (page 445) with the rest of the set. While resyncing, the member enters the [RECOVERING](#) (page 482) state.

Whenever the replica set replaces a *primary* in an election, the old primary's data collection may contain documents that did not have time to replicate to the *secondary* members. In this case the member rolls back those writes. During [rollback](#) (page 397), the member will have [RECOVERING](#) (page 482) state.

On secondaries, the `compact` and `replSetMaintenance` commands force the secondary to enter [RECOVERING](#) (page 482) state. This prevents clients from reading during those operations.

Error States Members in any error state can't vote.

FATAL

Members that encounter an unrecoverable error enter the [FATAL](#) (page 482) state. Members in this state requires administrator intervention.

UNKNOWN

Members that have never communicated status information to the replica set are in the [UNKNOWN](#) (page 482) state.

DOWN

Members that lose their connection to the replica set enter the [DOWN](#) (page 482) state.

SHUNNED

Members that are removed from the replica set enter the [SHUNNED](#) (page 482) state.

ROLLBACK

When a [SECONDARY](#) (page 481) rolls back a write operation after transitioning from [PRIMARY](#) (page 481), it enters the [ROLLBACK](#) (page 482) state. See [Rollbacks During Replica Set Failover](#) (page 397).

Read Preference Reference

Read preference describes how MongoDB clients route read operations to members of a *replica set*.

By default, an application directs its read operations to the *primary* member in a *replica set*. Reading from the primary guarantees that read operations reflect the latest version of a document. However, by distributing some or all reads to secondary members of the replica set, you can improve read throughput or reduce latency for an application that does not require fully up-to-date data.

Read Preference Mode	Description
primary (page 483)	Default mode. All operations read from the current replica set <i>primary</i> .
primaryPreferred (page 483)	In most situations, operations read from the <i>primary</i> but if it is unavailable, operations read from <i>secondary</i> members.
secondary (page 483)	All operations read from the <i>secondary</i> members of the replica set.
secondaryPreferred (page 484)	In most situations, operations read from <i>secondary</i> members but if no <i>secondary</i> members are available, operations read from the <i>primary</i> .
nearest (page 484)	Operations read from the <i>nearest</i> member of the <i>replica set</i> , irrespective of the member's type.

Read Preference Modes

primary

All read operations use only the current replica set *primary*. This is the default. If the primary is unavailable, read operations produce an error or throw an exception.

The [primary](#) (page 483) read preference mode is not compatible with read preference modes that use *tag sets* (page 403). If you specify a tag set with [primary](#) (page 483), the driver will produce an error.

primaryPreferred

In most situations, operations read from the *primary* member of the set. However, if the primary is unavailable, as is the case during *failover* situations, operations read from secondary members.

When the read preference includes a *tag set* (page 403), the client reads first from the primary, if available, and then from *secondaries* that match the specified tags. If no secondaries have matching tags, the read operation produces an error.

Since the application may receive data from a secondary, read operations using the [primaryPreferred](#) (page 483) mode may return stale data in some situations.

Warning: Changed in version 2.2: `mongos` added full support for read preferences.

When connecting to a `mongos` instance older than 2.2, using a client that supports read preference modes, [primaryPreferred](#) (page 483) will send queries to secondaries.

secondary

Operations read *only* from the *secondary* members of the set. If no secondaries are available, then this read operation produces an error or exception.

Most sets have at least one secondary, but there are situations where there may be no available secondary. For example, a set with a primary, a secondary, and an *arbiter* may not have any secondaries if a member is in recovering state or unavailable.

When the read preference includes a *tag set* (page 403), the client attempts to find secondary members that match the specified tag set and directs reads to a random secondary from among the [nearest group](#) (page 404).

If no secondaries have matching tags, the read operation produces an error.¹⁵

Read operations using the [secondary](#) (page 483) mode may return stale data.

secondaryPreferred

In most situations, operations read from *secondary* members, but in situations where the set consists of a single *primary* (and no other members,) the read operation will use the set's primary.

When the read preference includes a [tag set](#) (page 403), the client attempts to find a secondary member that matches the specified tag set and directs reads to a random secondary from among the [nearest group](#) (page 404). If no secondaries have matching tags, the client ignores tags and reads from the primary.

Read operations using the [secondaryPreferred](#) (page 484) mode may return stale data.

Warning: In some situations using [secondaryPreferred](#) (page 484) to distribute read load to replica sets may carry significant operational risk: if all secondaries are unavailable and your set has enough *arbiters* to prevent the primary from stepping down, then the primary will receive all traffic from clients.

For this reason, use [secondary](#) (page 483) to distribute read load to replica sets, not [secondaryPreferred](#) (page 484).

nearest

The driver reads from the *nearest* member of the *set* according to the [member selection](#) (page 404) process. Reads in the [nearest](#) (page 484) mode do not consider the member's *type*. Reads in [nearest](#) (page 484) mode may read from both primaries and secondaries.

Set this mode to minimize the effect of network latency on read operations without preference for current or stale data.

If you specify a [tag set](#) (page 403), the client attempts to find a replica set member that matches the specified tag set and directs reads to an arbitrary member from among the [nearest group](#) (page 404).

Read operations using the [nearest](#) (page 484) mode may return stale data.

Note: All operations read from a member of the nearest group of the replica set that matches the specified read preference mode. The [nearest](#) (page 484) mode prefers low latency reads over a member's *primary* or *secondary* status.

For [nearest](#) (page 484), the client assembles a list of acceptable hosts based on tag set and then narrows that list to the host with the shortest ping time and all other members of the set that are within the “local threshold,” or acceptable latency. See [Member Selection](#) (page 404) for more information.

Read Preferences for Database Commands

Because some *database commands* read and return data from the database, all of the official drivers support full [read preference mode semantics](#) (page 483) for the following commands:

- `group`
- `mapReduce`¹⁶
- `aggregate`
- `collStats`

¹⁵ If your set has more than one secondary, and you use the [secondary](#) (page 483) read preference mode, consider the following effect. If you have a [three member replica set](#) (page 387) with a primary and two secondaries, and if one secondary becomes unavailable, all [secondary](#) (page 483) queries must target the remaining secondary. This will double the load on this secondary. Plan and provide capacity to support this as needed.

¹⁶ Only “inline” mapReduce operations that do not write data support read preference, otherwise these operations must run on the *primary* members.

- dbStats
- count
- distinct
- geoNear
- geoSearch
- geoWalk

New in version 2.4: mongos adds support for routing commands to shards using read preferences. Previously mongos sent all commands to shards' primaries.

Sharding

Sharding is the process of storing data records across multiple machines and is MongoDB's approach to meeting the demands of data growth. As the size of the data increases, a single machine may not be sufficient to store the data nor provide an acceptable read and write throughput. Sharding solves the problem with horizontal scaling. With sharding, you add more machines to support data growth and the demands of read and write operations.

Sharding Introduction (page 487) A high-level introduction to horizontal scaling, data partitioning, and sharded clusters in MongoDB.

Sharding Concepts (page 492) The core documentation of sharded cluster features, configuration, architecture and behavior.

Sharded Cluster Components (page 493) A sharded cluster consists of shards, config servers, and mongos instances.

Sharded Cluster Architectures (page 497) Outlines the requirements for sharded clusters, and provides examples of several possible architectures for sharded clusters.

Sharded Cluster Behavior (page 499) Discusses the operations of sharded clusters with regards to the automatic balancing of data in a cluster and other related availability and security considerations.

Sharding Mechanics (page 507) Discusses the internal operation and behavior of sharded clusters, including chunk migration, balancing, and the cluster metadata.

Sharded Cluster Tutorials (page 513) Tutorials that describe common procedures and administrative operations relevant to the use and maintenance of sharded clusters.

Sharding Reference (page 554) Reference for sharding-related functions and operations.

9.1 Sharding Introduction

Sharding is a method for storing data across multiple machines. MongoDB uses sharding to support deployments with very large data sets and high throughput operations.

9.1.1 Purpose of Sharding

Database systems with large data sets and high throughput applications can challenge the capacity of a single server. High query rates can exhaust the CPU capacity of the server. Larger data sets exceed the storage capacity of a single machine. Finally, working set sizes larger than the system's RAM stress the I/O capacity of disk drives.

To address these issues of scales, database systems have two basic approaches: **vertical scaling** and **sharding**.

Vertical scaling adds more CPU and storage resources to increase capacity. Scaling by adding capacity has limitations: high performance systems with large numbers of CPUs and large amount of RAM are disproportionately *more expensive* than smaller systems. Additionally, cloud-based providers may only allow users to provision smaller instances. As a result there is a *practical maximum* capability for vertical scaling.

Sharding, or *horizontal scaling*, by contrast, divides the data set and distributes the data over multiple servers, or **shards**. Each shard is an independent database, and collectively, the shards make up a single logical database.

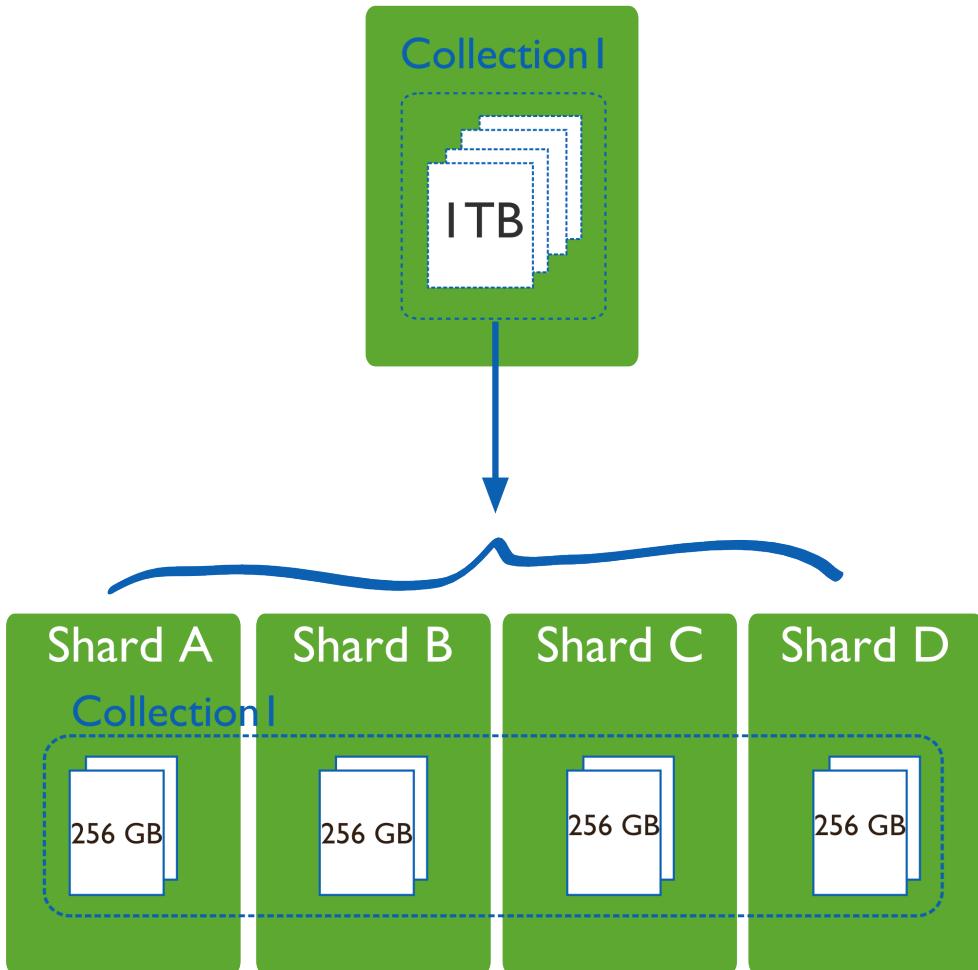


Figure 9.1: Diagram of a large collection with data distributed across 4 shards.

Sharding addresses the challenge of scaling to support high throughput and large data sets:

- Sharding reduces the number of operations each shard handles. Each shard processes fewer operations as the cluster grows. As a result, shared clusters can increase capacity and throughput *horizontally*.

For example, to insert data, the application only needs to access the shard responsible for that records.

- Sharding reduces the amount of data that each server needs to store. Each shard stores less data as the cluster grows.

For example, if a database has a 1 terabyte data set, and there are 4 shards, then each shard might hold only 256GB of data. If there are 40 shards, then each shard might hold only 25GB of data.

9.1.2 Sharding in MongoDB

MongoDB supports sharding through the configuration of a *sharded clusters*.

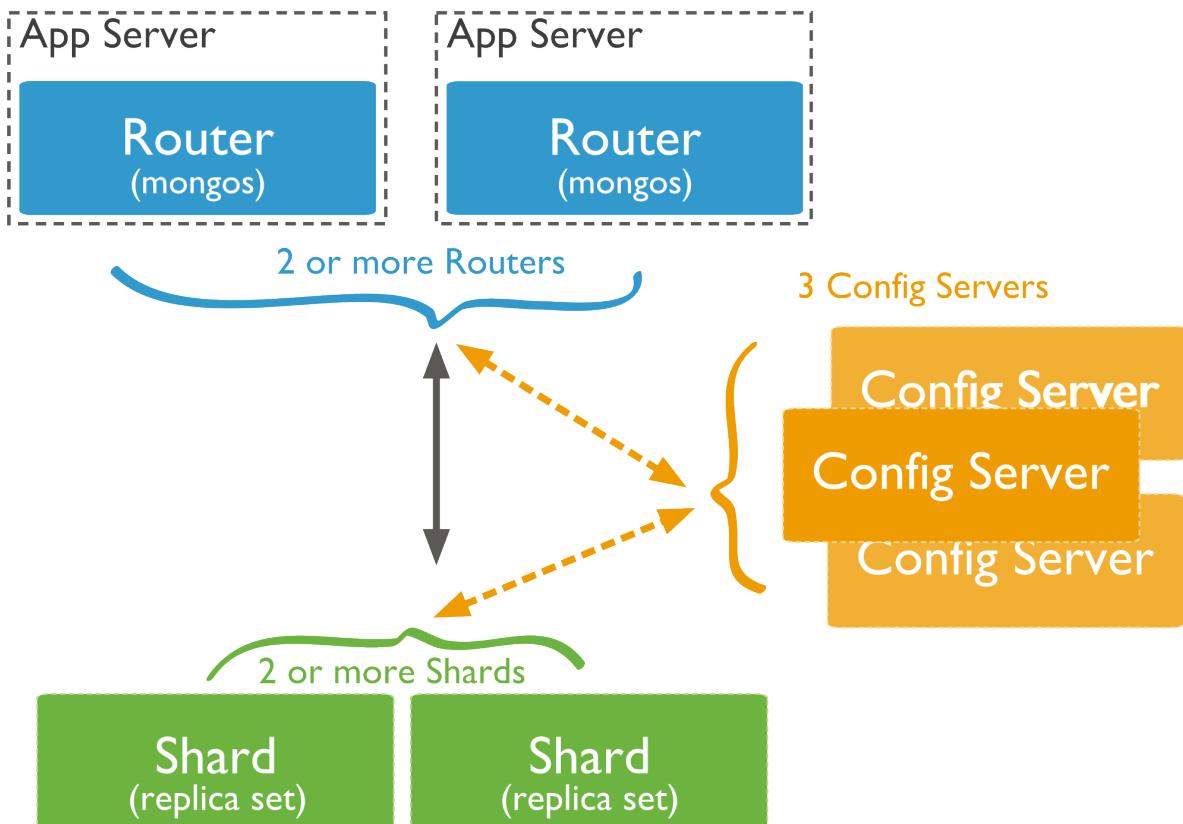


Figure 9.2: Diagram of a sample sharded cluster for production purposes. Contains exactly 3 config servers, 2 or more mongos query routers, and at least 2 shards. The shards are replica sets.

Sharded cluster has the following components: *shards*, *query routers* and *config servers*.

Shards store the data. To provide high availability and data consistency, in a production sharded cluster, each shard is a *replica set*¹. For more information on replica sets, see [Replica Sets](#) (page 377).

Query Routers, or `mongos` instances, interface with client applications and direct operations to the appropriate shard or shards. The query router processes and targets operations to shards and then returns results to the clients. A sharded cluster can contain more than one query router to divide the client request load. A client sends requests to one query router. Most sharded cluster have many query routers.

Config servers store the cluster's metadata. This data contains a mapping of the cluster's data set to the shards. The query router uses this metadata to target operations to specific shards. Production sharded clusters have *exactly* 3 config servers.

9.1.3 Data Partitioning

MongoDB distributes data, or shards, at the collection level. Sharding partitions a collection's data by the **shard key**.

¹ For development and testing purposes only, each **shard** can be a single `mongod` instead of a replica set. Do **not** deploy production clusters without 3 config servers.

Shard Keys

To shard a collection, you need to select a **shard key**. A *shard key* is either an indexed field or an indexed compound field that exists in every document in the collection. MongoDB divides the shard key values into **chunks** and distributes the *chunks* evenly across the shards. To divide the shard key values into chunks, MongoDB uses either **range based partitioning** and **hash based partitioning**. See [Shard Keys](#) (page 499) for more information.

Range Based Sharding

For *range-based sharding*, MongoDB divides the data set into ranges determined by the shard key values to provide **range based partitioning**. Consider a numeric shard key: If you visualize a number line that goes from negative infinity to positive infinity, each value of the shard key falls at some point on that line. MongoDB partitions this line into smaller, non-overlapping ranges called **chunks** where a chunk is range of values from some minimum value to some maximum value.

Given a range based partitioning system, documents with “close” shard key values are likely to be in the same chunk, and therefore on the same shard.

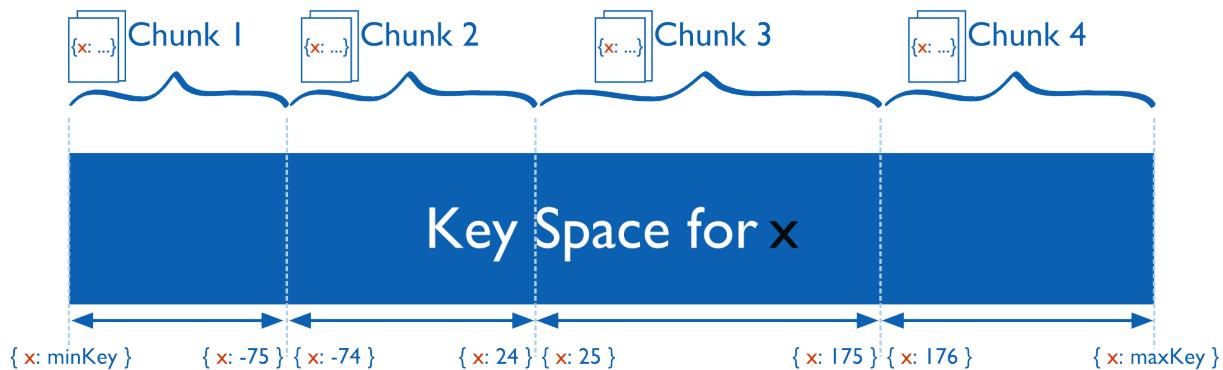


Figure 9.3: Diagram of the shard key value space segmented into smaller ranges or chunks.

Hash Based Sharding

For *hash based partitioning*, MongoDB computes a hash of a field’s value, and then uses these hashes to create chunks. With hash based partitioning, two documents with “close” shard key values are *unlikely* to be part of the same chunk. This ensures a more random distribution of a collection in the cluster.

Performance Distinctions between Range and Hash Based Partitioning

Range based partitioning supports more efficient range queries. Given a range query on the shard key, the query router can easily determine which chunks overlap that range and route the query to only those shards that contain these chunks.

However, range based partitioning can result in an uneven distribution of data, which may negate some of the benefits of sharding. For example, if the shard key is a linearly increasing field, such as time, then all requests for a given time range will map to the same chunk, and thus the same shard. In this situation, a small set of shards may receive the majority of requests and the system would not scale very well.

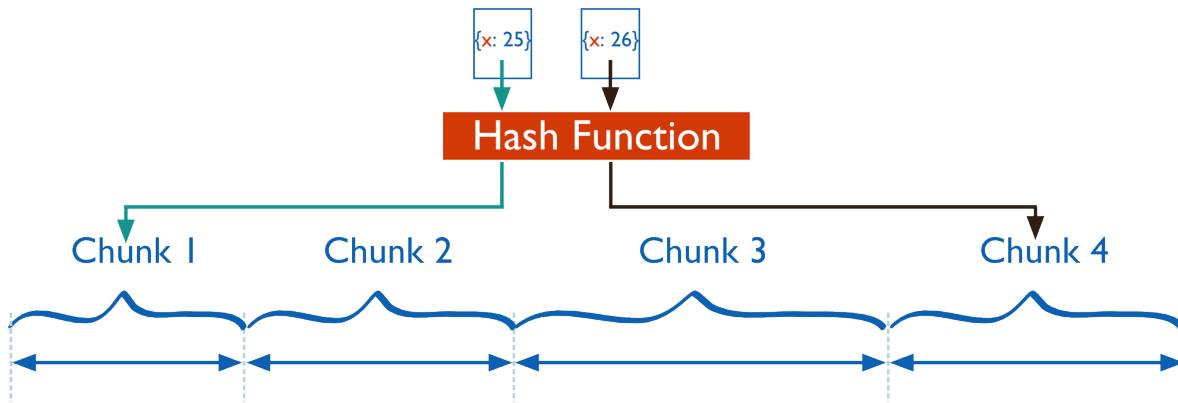


Figure 9.4: Diagram of the hashed based segmentation.

Hash based partitioning, by contrast, ensures an even distribution of data at the expense of efficient range queries. Hashed key values results in random distribution of data across chunks and therefore shards. But random distribution makes it more likely that a range query on the shard key will not be able to target a few shards but would more likely query every shard in order to return a result.

9.1.4 Maintaining a Balanced Data Distribution

The addition of new data or the addition of new servers can result in data distribution imbalances within the cluster, such as a particular shard contains significantly more chunks than another shard or a size of a chunk is significantly greater than other chunk sizes.

MongoDB ensures a balanced cluster using two background processes: splitting and the balancer.

Splitting

Splitting is a background process that keeps chunks from growing too large. When a chunk grows beyond a [specified chunk size](#) (page 511), MongoDB splits the chunk in half. Inserts and updates triggers splits. Splits are an efficient meta-data change. To create splits, MongoDB does *not* migrate any data or affect the shards.

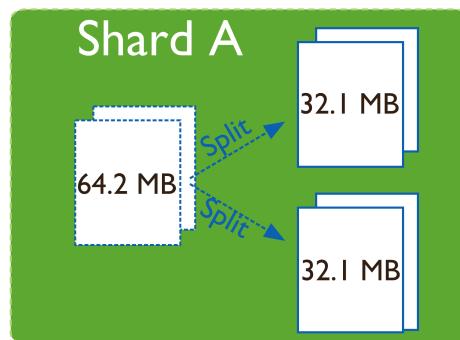


Figure 9.5: Diagram of a shard with a chunk that exceeds the default [chunk size](#) (page 511) of 64 MB and triggers a split of the chunk into two chunks.

Balancing

The [balancer](#) (page 508) is a background process that manages chunk migrations. The balancer runs in all of the query routers in a cluster.

When the distribution of a sharded collection in a cluster is uneven, the balancer process migrates chunks from the shard that has the largest number of chunks to the shard with the least number of chunks until the collection balances. For example: if collection `users` has 100 chunks on *shard 1* and 50 chunks on *shard 2*, the balancer will migrate chunks from *shard 1* to *shard 2* until the collection achieves balance.

The shards manage *chunk migrations* as a background operation. During migration, all requests for a chunk's data address the “origin” shard.

In a chunk migration, the *destination shard* receives all the documents in the chunk from the *origin shard*. Then, the destination shard captures and applies all changes made to the data during migration process. Finally, the destination shard updates the metadata regarding the location of the on *config server*.

If there’s an error during the migration, the balancer aborts the process leaving the chunk on the origin shard. MongoDB removes the chunks data from the origin shard **after** the migration completes successfully.

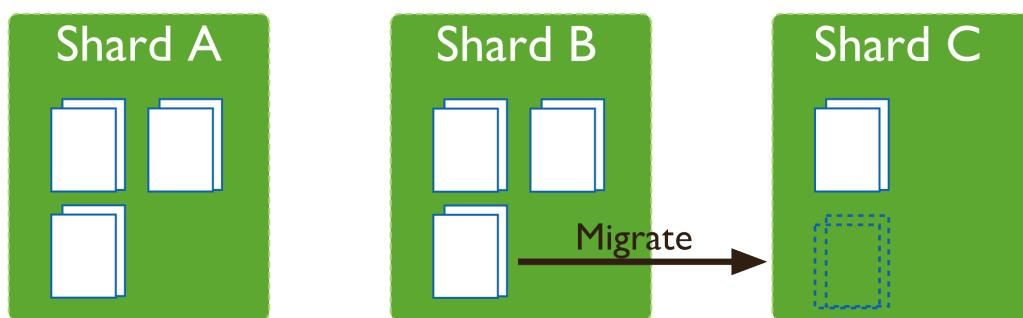


Figure 9.6: Diagram of a collection distributed across three shards. For this collection, the difference in the number of chunks between the shards reaches the [migration thresholds](#) (page 509) (in this case, 2) and triggers migration.

Adding and Removing Shards from the Cluster

Adding a shard to a cluster creates an imbalance since the new shard has no chunks. While MongoDB begins migrating data to the new shard immediately, it can take some time before the cluster balances.

When removing a shard, the balancer migrates all chunks from to other shards. After migrating all data and updating the meta data, you can safely remove the shard.

9.2 Sharding Concepts

These documents present the details of sharding in MongoDB. These include the components, the architectures, and the behaviors of MongoDB sharded clusters. For an overview of sharding and sharded clusters, see [Sharding Introduction](#) (page 487).

Sharded Cluster Components (page 493) A sharded cluster consists of shards, config servers, and `mongos` instances.

Shards (page 493) A shard is a `mongod` instance that holds a part of the sharded collection’s data.

[Config Servers \(page 496\)](#) Config servers hold the metadata about the cluster, such as the shard location of the data.

[Sharded Cluster Architectures \(page 497\)](#) Outlines the requirements for sharded clusters, and provides examples of several possible architectures for sharded clusters.

[Sharded Cluster Requirements \(page 497\)](#) Discusses the requirements for sharded clusters in MongoDB.

[Production Cluster Architecture \(page 498\)](#) Sharded cluster for production has component requirements to provide redundancy and high availability.

[Sharded Cluster Behavior \(page 499\)](#) Discusses the operations of sharded clusters with regards to the automatic balancing of data in a cluster and other related availability and security considerations.

[Shard Keys \(page 499\)](#) MongoDB uses the shard key to divide a collection's data across the cluster's shards.

[Sharded Cluster High Availability \(page 502\)](#) Sharded clusters provide ways to address some availability concerns.

[Sharded Cluster Query Routing \(page 503\)](#) The cluster's routers, or `mongos` instances, send reads and writes to the relevant shard or shards.

[Sharding Mechanics \(page 507\)](#) Discusses the internal operation and behavior of sharded clusters, including chunk migration, balancing, and the cluster metadata.

[Sharded Collection Balancing \(page 508\)](#) Balancing distributes a sharded collection's data cluster to all of the shards.

[Sharded Cluster Metadata \(page 512\)](#) The cluster maintains internal metadata that reflects the location of data within the cluster.

9.2.1 Sharded Cluster Components

Sharded clusters implement *sharding*. A sharded cluster consists of the following components:

Shards A shard is a MongoDB instance that holds a subset of a collection's data. Each shard is either a single `mongod` instance or a *replica set*. In production, all shards are replica sets. For more information see [Shards \(page 493\)](#).

Config Servers Each *config server* (page 496) is a `mongod` instance that holds metadata about the cluster. The metadata maps *chunks* to shards. For more information, see [Config Servers \(page 496\)](#).

Routing Instances Each router is a `mongos` instance that routes the reads and writes from applications to the shards. Applications do not access the shards directly. For more information see [Sharded Cluster Query Routing \(page 503\)](#).

Enable sharding in MongoDB on a per-collection basis. For each collection you shard, you will specify a *shard key* for that collection.

Deploy a sharded cluster, see [Deploy a Sharded Cluster \(page 514\)](#).

Shards

A shard is a *replica set* or a single `mongod` that contains a subset of the data for the sharded cluster. Together, the cluster's shards hold the entire data set for the cluster.

Typically each shard is a replica set. The replica set provides redundancy and high availability for the data in each shard.

Important: MongoDB shards data on a *per collection* basis. You *must* access all data in a sharded cluster via the `mongos` instances. If you connect directly to a shard, you will see only its fraction of the cluster's data. There is no

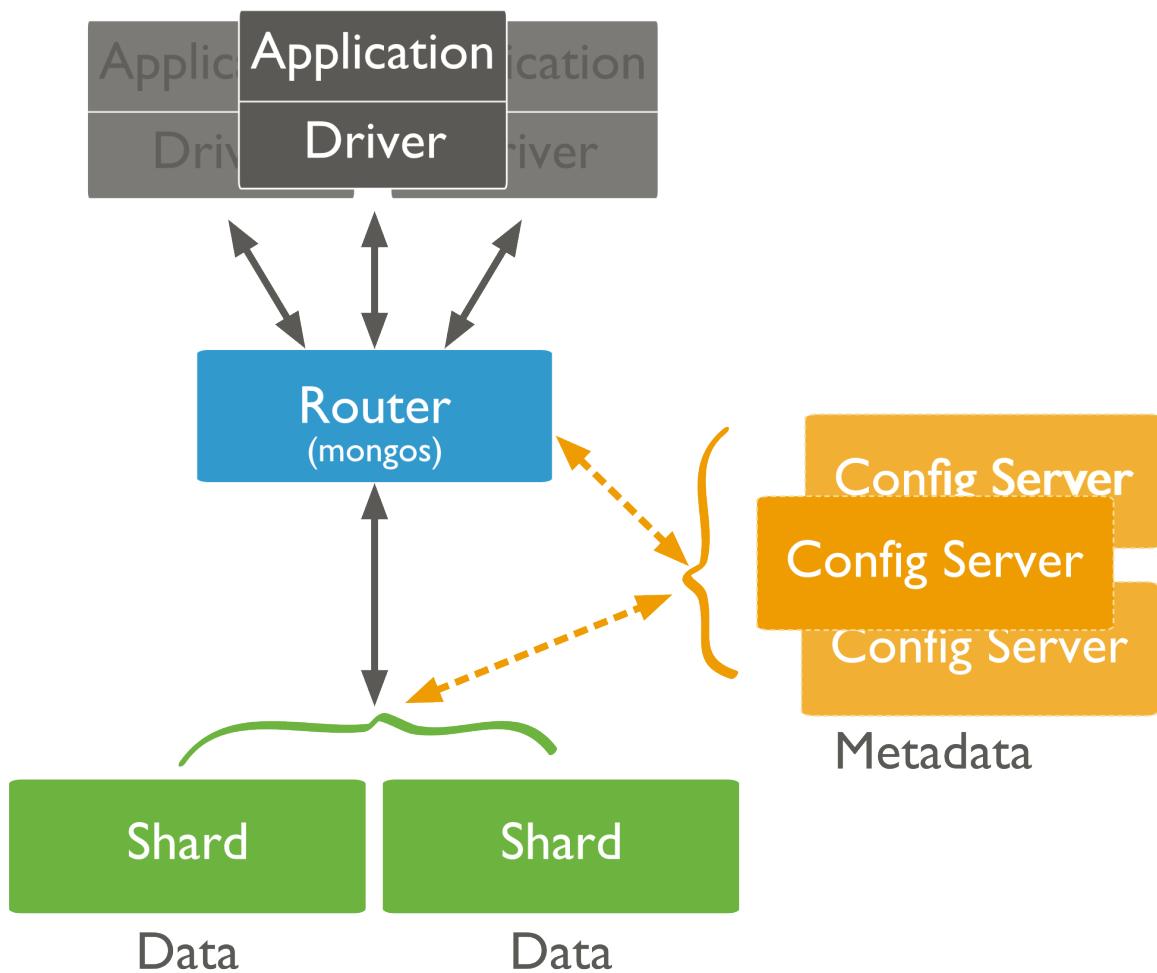


Figure 9.7: Diagram of a sharded cluster.

particular order to the data set on a specific shard. MongoDB does not guarantee that any two contiguous chunks will reside on a single shard.

Primary Shard

Every database has a “primary”² shard that holds all the un-sharded collections in that database.

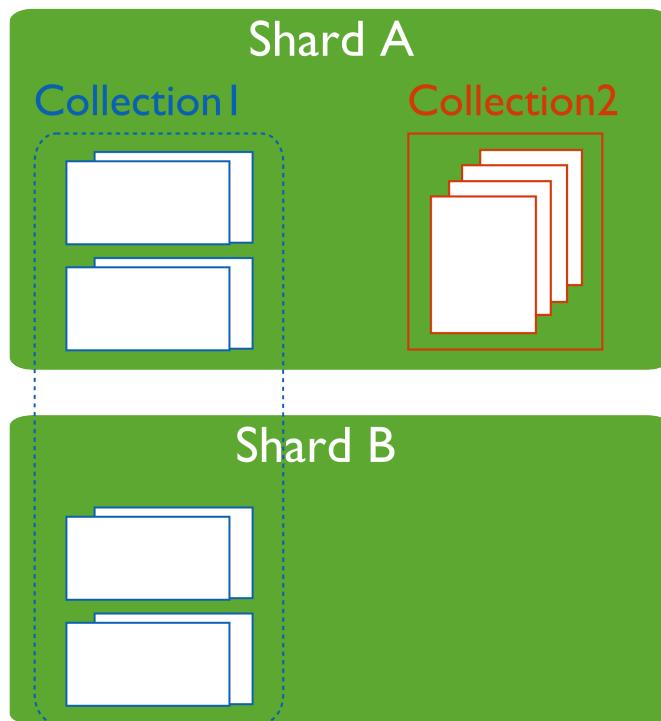


Figure 9.8: Diagram of a primary shard. A primary shard contains non-sharded collections as well as chunks of documents from sharded collections. Shard A is the primary shard.

To change the primary shard for a database, use the `movePrimary` command.

Warning: The `movePrimary` command can be expensive because it copies all non-sharded data to the new shard. During this time, this data will be unavailable for other operations.

When you deploy a new *sharded cluster*, the “first” shard becomes the primary shard for all existing databases before enabling sharding. Databases created subsequently may reside on any shard in the cluster.

Shard Status

Use the `sh.status()` method in the `mongo` shell to see an overview of the cluster. This reports includes which shard is primary for the database and the *chunk* distribution across the shards. See `sh.status()` method for more details.

² The term “primary” shard has nothing to do with the term *primary* in the context of *replica sets*.

Config Servers

Config servers are special mongod instances that store the [metadata](#) (page 512) for a sharded cluster. Config servers use a two-phase commit to ensure immediate consistency and reliability. Config servers *do not* run as replica sets. All config servers must be available to deploy a sharded cluster or to make any changes to cluster metadata.

A production sharded cluster has *exactly three* config servers. For testing purposes you may deploy a cluster with a single config server. But to ensure redundancy and safety in production, you should always use three.

Warning: If your cluster has a single config server, then the config server is a single point of failure. If the config server is inaccessible, the cluster is not accessible. If you cannot recover the data on a config server, the cluster will be inoperable.

Always use three config servers for production deployments.

Config servers store metadata for a single sharded cluster. Each cluster must have its own 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.

Config Database

Config servers store the metadata in the [config database](#) (page 555). The mongos instances cache this data and use it to route reads and writes to shards.

Read and Write Operations on Config Servers

MongoDB only writes data to the config server in the following cases:

- To create splits in existing chunks. For more information, see [chunk splitting](#) (page 511).
- To migrate a chunk between shards. For more information, see [chunk migration](#) (page 510).

MongoDB reads data from the config server data in the following cases:

- A new mongos starts for the first time, or an existing mongos restarts.
- After a chunk migration, the mongos instances update themselves with the new cluster metadata.

MongoDB also uses the config server to manage distributed locks.

Config Server Availability

If one or two config servers become unavailable, the cluster's metadata becomes *read only*. You can still read and write data from the shards, but no chunk migrations or splits will occur until all three servers are available.

If all three config servers are unavailable, you can still use the cluster if you do not restart the mongos instances until after the config servers are accessible again. If you restart the mongos instances before the config servers are available, the mongos will be unable to route reads and writes.

Clusters become inoperable without the cluster metadata. *Always*, ensure that the config servers remain available and intact. As such, backups of config servers are critical. The data on the config server is small compared to the data stored in a cluster. This means the config server has a relatively low activity load, and the config server does not need to be always available to support a sharded cluster. As a result, it is easy to back up the config servers.

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 [Renaming Config Servers and Cluster Availability](#) (page 502) for more information.

9.2.2 Sharded Cluster Architectures

The following documents introduce deployment patterns for sharded clusters.

[Sharded Cluster Requirements](#) (page 497) Discusses the requirements for sharded clusters in MongoDB.

[Production Cluster Architecture](#) (page 498) Sharded cluster for production has component requirements to provide redundancy and high availability.

[Sharded Cluster Test Architecture](#) (page 498) Sharded clusters for testing and development can have fewer components.

Sharded Cluster Requirements

While sharding is a powerful and compelling feature, sharded clusters have significant infrastructure requirements and increases the overall complexity of a deployment. As a result, only deploy sharded clusters when indicated by application and operational requirements.

Sharding is the *only* solution for some classes of deployments. Use *sharded clusters* if:

- your data set approaches or exceeds the storage capacity of a single MongoDB instance.
- the size of your system's active *working set* will soon exceed the capacity of your system's *maximum RAM*.
- a single MongoDB instance cannot meet the demands of your write operations, and all other approaches have not reduced contention.

If these attributes are not present in your system, sharding will only add complexity to your system without adding much benefit.

Important: It takes time and resources to deploy sharding. If your system has *already* reached or exceeded its capacity, it will be difficult to deploy sharding without impacting your application.

As a result, if you think you will need to partition your database in the future, **do not** wait until your system is overcapacity to enable sharding.

When designing your data model, take into consideration your sharding needs.

Data Quantity Requirements

Your cluster should manage a large quantity of data if sharding is to have an effect. The default *chunk* size is 64 megabytes. And the [balancer](#) (page 508) will not begin moving data across shards until the imbalance of chunks among the shards exceeds the [migration threshold](#) (page 509). In practical terms, unless your cluster has many hundreds of megabytes of data, your data will remain on a single shard.

In some situations, you may need to shard a small collection of data. But most of the time, sharding a small collection is not worth the added complexity and overhead unless you need additional write capacity. If you have a small data set, a properly configured single MongoDB instance or a replica set will usually be enough for your persistence layer needs.

Chunk size is user configurable. For most deployments, the default value is of 64 megabytes is ideal. See [Chunk Size](#) (page 511) for more information.

Production Cluster Architecture

In a production cluster, you must ensure that data is redundant and that your systems are highly available. To that end, a production cluster must have the following components:

- Three [config servers](#) (page 496). Each config servers must be on separate machines. A single *sharded cluster* must have exclusive use of its [config servers](#) (page 496). If you have multiple sharded clusters, you will need to have a group of config servers for each cluster.
- Two or more *replica sets*. These replica sets are the *shards*. For information on replica sets, see [Replication](#) (page 373).
- One or more `mongos` instances. `mongos` is the routers for the cluster. Typically, deployments have one `mongos` instance on each application server. You may also may deploy a group of `mongos` instances and use a proxy/load balancer between the application and the `mongos`.

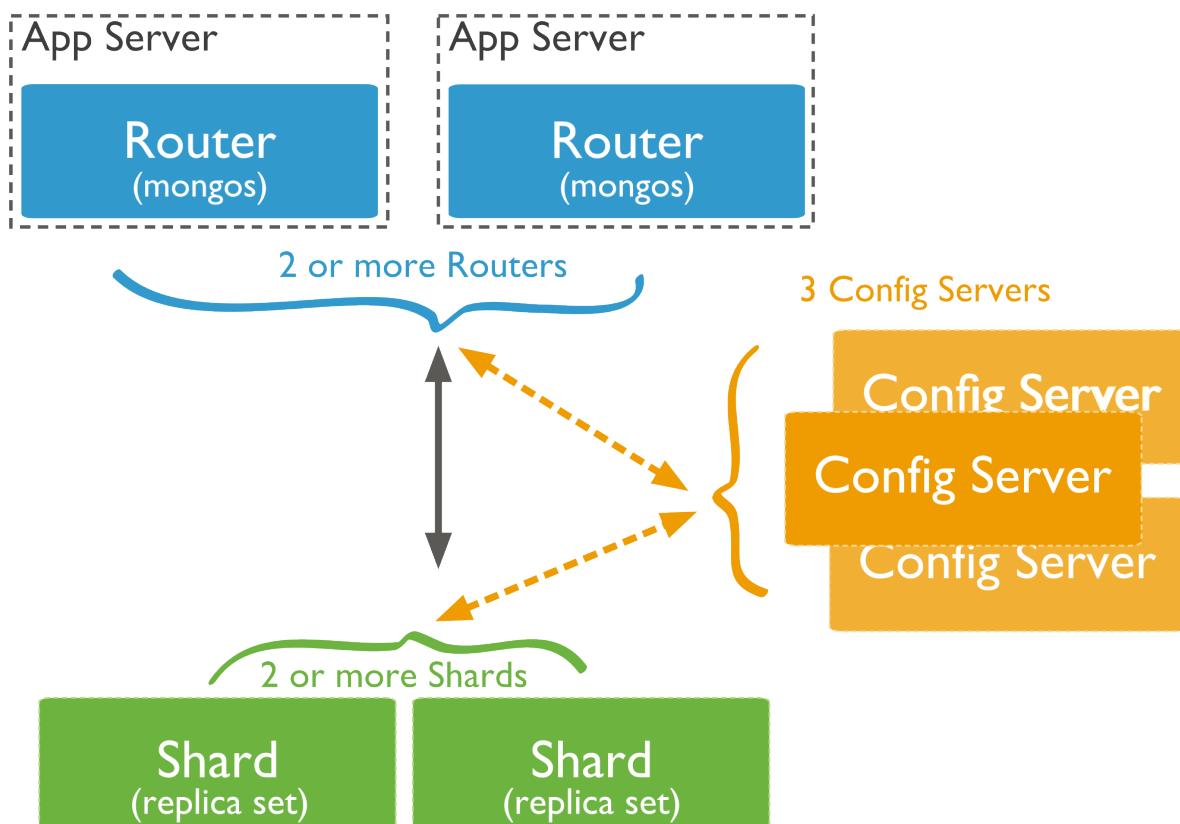


Figure 9.9: Diagram of a sample sharded cluster for production purposes. Contains exactly 3 config servers, 2 or more `mongos` query routers, and at least 2 shards. The shards are replica sets.

Sharded Cluster Test Architecture

Warning: Use the test cluster architecture for testing and development only.

For testing and development, you can deploy a minimal sharded clusters cluster. These **non-production** clusters have the following components:

- One [config server](#) (page 496).
- At least one shard. Shards are either *replica sets* or a standalone `mongod` instances.
- One `mongos` instance.

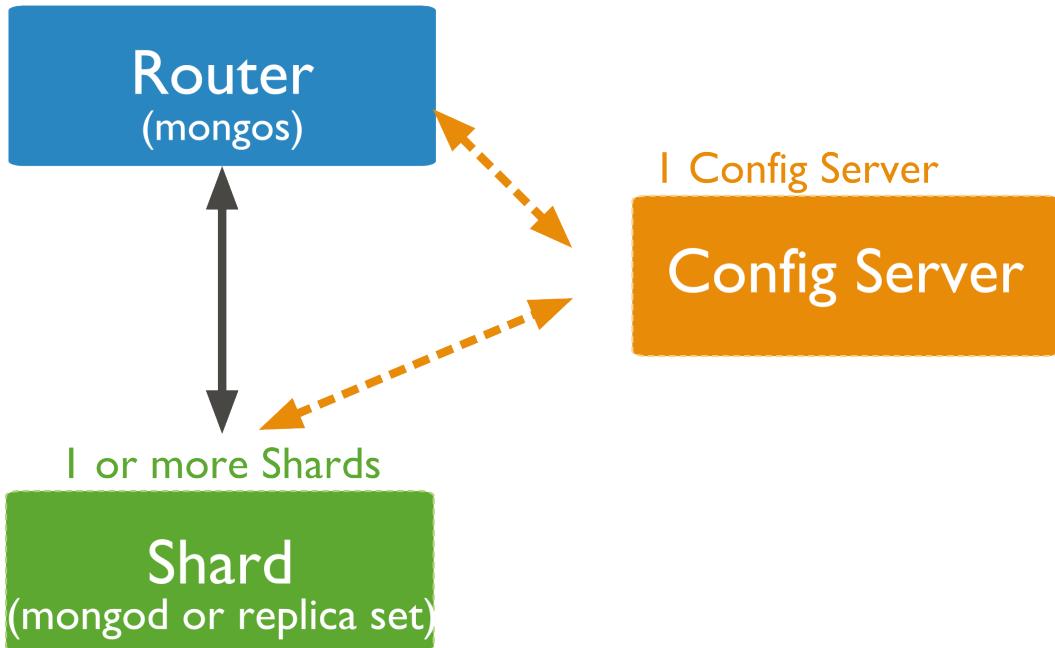


Figure 9.10: Diagram of a sample sharded cluster for testing/development purposes only. Contains only 1 config server, 1 `mongos` router, and at least 1 shard. The shard can be either a replica set or a standalone `mongod` instance.

See

[Production Cluster Architecture](#) (page 498)

9.2.3 Sharded Cluster Behavior

These documents address the distribution of data and queries to a sharded cluster as well as specific security and availability considerations for sharded clusters.

[Shard Keys \(page 499\)](#) MongoDB uses the shard key to divide a collection's data across the cluster's shards.

[Sharded Cluster High Availability \(page 502\)](#) Sharded clusters provide ways to address some availability concerns.

[Sharded Cluster Query Routing \(page 503\)](#) The cluster's routers, or `mongos` instances, send reads and writes to the relevant shard or shards.

Shard Keys

The shard key determines the distribution of the collection's *documents* among the cluster's *shards*. The shard key is either an indexed *field* or an indexed compound field that exists in every document in the collection.

MongoDB partitions data in the collection using ranges of shard key values. Each range, or *chunk*, defines a non-overlapping range of shard key values. MongoDB distributes the chunks, and their documents, among the shards in the cluster.

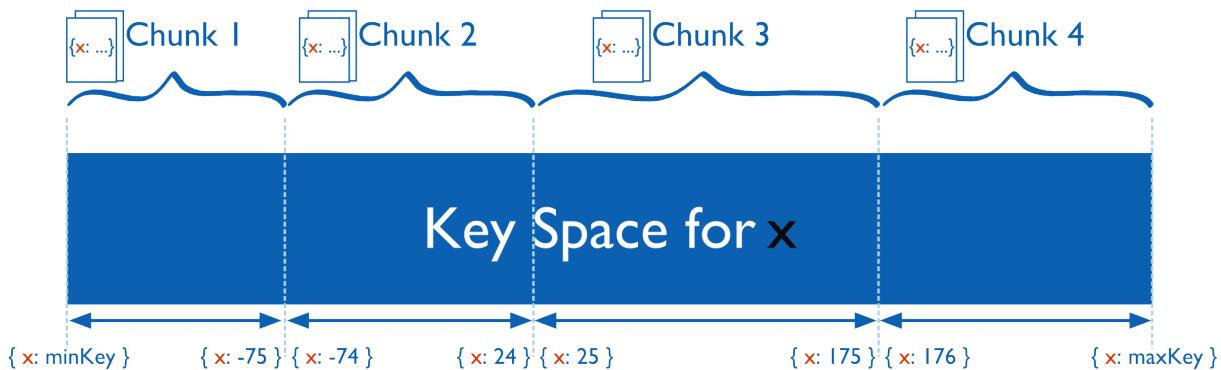


Figure 9.11: Diagram of the shard key value space segmented into smaller ranges or chunks.

When a chunk grows beyond the [chunk size](#) (page 511), MongoDB *splits* the chunk into smaller chunks, always based on ranges in the shard key.

Important: Shard keys are immutable and cannot be changed after insertion. See the [system limits for sharded cluster](#) for more information.

Hashed Shard Keys

New in version 2.4.

Hashed shard keys use a [hashed index](#) (page 338) of a single field as the *shard key* to partition data across your sharded cluster.

The field you choose as your hashed shard key should have a good cardinality, or large number of different values. Hashed keys work well with fields that increase monotonically like *ObjectId* values or timestamps.

If you shard an empty collection using a hashed shard key, MongoDB will automatically create and migrate chunks so that each shard has two chunks. You can control how many chunks MongoDB will create with the `numInitialChunks` parameter to `shardCollection` or by manually creating chunks on the empty collection using the `split` command.

To shard a collection using a hashed shard key, see [Shard a Collection Using a Hashed Shard Key](#) (page 520).

Tip

MongoDB automatically computes the hashes when resolving queries using hashed indexes. Applications do **not** need to compute hashes.

Impacts of Shard Keys on Cluster Operations

The shard key affects write and query performance by determining how the MongoDB partitions data in the cluster and how effectively the mongos instances can direct operations to the cluster. Consider the following operational impacts of shard key selection:

Write Scaling Some possible shard keys will allow your application to take advantage of the increased write capacity that the cluster can provide, while others do not. Consider the following example where you shard by the values of the default `_id` field, which is *ObjectID*.

MongoDB generates `ObjectID` values upon document creation to produce a unique identifier for the object. However, the most significant bits of data in this value represent a time stamp, which means that they increment in a regular and predictable pattern. Even though this value has *high cardinality* (page 519), when using this, *any date, or other monotonically increasing number* as the shard key, all insert operations will be storing data into a single chunk, and therefore, a single shard. As a result, the write capacity of this shard will define the effective write capacity of the cluster.

A shard key that increases monotonically will not hinder performance if you have a very low insert rate, or if most of your write operations are `update()` operations distributed through your entire data set. Generally, choose shard keys that have *both* high cardinality and will distribute write operations across the *entire cluster*.

Typically, a computed shard key that has some amount of “randomness,” such as ones that include a cryptographic hash (i.e. MD5 or SHA1) of other content in the document, will allow the cluster to scale write operations. However, random shard keys do not typically provide *query isolation* (page 501), which is another important characteristic of shard keys.

New in version 2.4: MongoDB makes it possible to shard a collection on a hashed index. This can greatly improve write scaling. See [Shard a Collection Using a Hashed Shard Key](#) (page 520).

Querying The `mongos` provides an interface for applications to interact with sharded clusters that hides the complexity of *data partitioning*. A `mongos` receives queries from applications, and uses metadata from the [config server](#) (page 496), to route queries to the `mongod` instances with the appropriate data. While the `mongos` succeeds in making all querying operational in sharded environments, the *shard key* you select can have a profound affect on query performance.

See also:

The [Sharded Cluster Query Routing](#) (page 503) and [config server](#) (page 496) sections for a more general overview of querying in sharded environments.

Query Isolation The fastest queries in a sharded environment are those that `mongos` will route to a single shard, using the *shard key* and the cluster meta data from the [config server](#) (page 496). For queries that don’t include the shard key, `mongos` must query all shards, wait for their response and then return the result to the application. These “scatter/gather” queries can be long running operations.

If your query includes the first component of a compound shard key ³, the `mongos` can route the query directly to a single shard, or a small number of shards, which provides better performance. Even if you query values of the shard key reside in different chunks, the `mongos` will route queries directly to specific shards.

To select a shard key for a collection:

- determine the most commonly included fields in queries for a given application
- find which of these operations are most performance dependent.

If this field has low cardinality (i.e not sufficiently selective) you should add a second field to the shard key making a compound shard key. The data may become more splittable with a compound shard key.

See

[Sharded Cluster Query Routing](#) (page 503) for more information on query operations in the context of sharded clusters.

³ In many ways, you can think of the shard key a cluster-wide unique index. However, be aware that sharded systems cannot enforce cluster-wide unique indexes *unless* the unique field is in the shard key. Consider the [Index Concepts](#) (page 314) page for more information on indexes and compound indexes.

Sorting In sharded systems, the `mongos` performs a merge-sort of all sorted query results from the shards. See [Sharded Cluster Query Routing](#) (page 503) and [Use Indexes to Sort Query Results](#) (page 366) for more information.

Sharded Cluster High Availability

A [production](#) (page 498) *cluster* has no single point of failure. This section introduces the availability concerns for MongoDB deployments in general and highlights potential failure scenarios and available resolutions.

Application Servers or `mongos` Instances Become Unavailable

If each application server has its own `mongos` instance, other application servers can continue access the database. Furthermore, `mongos` instances do not maintain persistent state, and they can restart and become unavailable without loosing any state or data. When a `mongos` instance starts, it retrieves a copy of the *config database* and can begin routing queries.

A Single `mongod` Becomes Unavailable in a Shard

[Replica sets](#) (page 373) provide high availability for shards. If the unavailable `mongod` is a *primary*, then the replica set will [elect](#) (page 393) a new primary. If the unavailable `mongod` is a *secondary*, and it disconnects the primary and secondary will continue to hold all data. In a three member replica set, even if a single member of the set experiences catastrophic failure, two other members have full copies of the data.⁴

Always investigate availability interruptions and failures. If a system is unrecoverable, replace it and create a new member of the replica set as soon as possible to replace the lost redundancy.

All Members of a Replica Set Become Unavailable

If all members of a replica set within a shard are unavailable, all data held in that shard is unavailable. However, the data on all other shards will remain available, and it's possible to read and write data to the other shards. However, your application must be able to deal with partial results, and you should investigate the cause of the interruption and attempt to recover the shard as soon as possible.

One or Two Config Databases Become Unavailable

Three distinct `mongod` instances provide the *config database* using a special two-phase commits to maintain consistent state between these `mongod` instances. Cluster operation will continue as normal but [chunk migration](#) (page 508) and the cluster can create no new [chunk splits](#) (page 545). Replace the config server as soon as possible. If all multiple config databases become unavailable, the cluster can become inoperable.

Note: All config servers must be running and available when you first initiate a *sharded cluster*.

Renaming Config Servers and Cluster Availability

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.

⁴ If an unavailable secondary becomes available while it still has current oplog entries, it can catch up to the latest state of the set using the normal *replication process*, otherwise it must perform an *initial sync*.

To avoid downtime when renaming config servers, use DNS names unrelated to physical or virtual hostnames to refer to your [config servers](#) (page 496).

Generally, refer to each config server using a name in DNS (e.g. a CNAME record). When specifying the config server connection string to mongos, DNS use this name. These records make it possible to renumber or rename config servers without changing the connection string and without having to restart the entire cluster.

Shard Keys and Cluster Availability

The most important consideration when choosing a *shard key* are:

- to ensure that MongoDB will be able to distribute data evenly among shards, and
- to scale writes across the cluster, and
- to ensure that mongos can isolate most queries to a specific mongod.

Furthermore:

- Each shard should be a *replica set*, if a specific mongod instance fails, the replica set members will elect another to be *primary* and continue operation. However, if an entire shard is unreachable or fails for some reason, that data will be unavailable.
- If the shard key allows the mongos to isolate most operations to a single shard, then the failure of a single shard will only render *some* data unavailable.
- If your shard key distributes data required for every operation throughout the cluster, then the failure of the entire shard will render the entire cluster unavailable.

In essence, this concern for reliability simply underscores the importance of choosing a shard key that isolates query operations to a single shard.

Sharded Cluster Query Routing

MongoDB mongos instances route queries and write operations to *shards* in a sharded cluster. mongos provide the only interface to a sharded cluster from the perspective of applications. Applications never connect or communicate directly with the shards.

The mongos tracks what data is on which shard by caching the metadata from the [config servers](#) (page 496). The mongos uses the metadata to route operations from applications and clients to the mongod instances. A mongos has no *persistent* state and consumes minimal system resources.

The most common practice is to run mongos instances on the same systems as your application servers, but you can maintain mongos instances on the shards or on other dedicated resources.

Note: Changed in version 2.1.

Some aggregation operations using the aggregate command (i.e. db.collection.aggregate()) will cause mongos instances to require more CPU resources than in previous versions. This modified performance profile may dictate alternate architecture decisions if you use the *aggregation framework* extensively in a sharded environment.

Routing Process

A mongos instance uses the following processes to route queries and return results.

How mongos Determines which Shards Receive a Query A `mongos` instance routes a query to a *cluster* by:

1. Determining the list of *shards* that must receive the query.
2. Establishing a cursor on all targeted shards.

In some cases, when the *shard key* or a prefix of the shard key is a part of the query, the `mongos` can route the query to a subset of the shards. Otherwise, the `mongos` must direct the query to *all* shards that hold documents for that collection.

Example

Given the following shard key:

```
{ zipcode: 1, u_id: 1, c_date: 1 }
```

Depending on the distribution of chunks in the cluster, the `mongos` may be able to target the query at a subset of shards, if the query contains the following fields:

```
{ zipcode: 1 }
{ zipcode: 1, u_id: 1 }
{ zipcode: 1, u_id: 1, c_date: 1 }
```

How mongos Handles Query Modifiers If the result of the query is not sorted, the `mongos` instance opens a result cursor that “round robins” results from all cursors on the shards.

Changed in version 2.0.5: In versions prior to 2.0.5, the `mongos` exhausted each cursor, one by one.

If the query specifies sorted results using the `sort()` cursor method, the `mongos` instance passes the `$orderby` option to the shards. When the `mongos` receives results it performs an incremental *merge sort* of the results while returning them to the client.

If the query limits the size of the result set using the `limit()` cursor method, the `mongos` instance passes that limit to the shards and then re-applies the limit to the result before returning the result to the client.

If the query specifies a number of records to *skip* using the `skip()` cursor method, the `mongos` *cannot* pass the skip to the shards, but rather retrieves unskipped results from the shards and skips the appropriate number of documents when assembling the complete result. However, when used in conjunction with a `limit()`, the `mongos` will pass the `limit` plus the value of the `skip()` to the shards to improve the efficiency of these operations.

Detect Connections to mongos Instances

To detect if the MongoDB instance that your client is connected to is `mongos`, use the `isMaster` command. When a client connects to a `mongos`, `isMaster` returns a document with a `msg` field that holds the string `isdbgrid`. For example:

```
{
  "ismaster" : true,
  "msg" : "isdbgrid",
  "maxBsonObjectSize" : 16777216,
  "ok" : 1
}
```

If the application is instead connected to a `mongod`, the returned document does not include the `isdbgrid` string.

Broadcast Operations and Targeted Operations

In general, operations in a sharded environment are either:

- Broadcast to all shards in the cluster that hold documents in a collection
- Targeted at a single shard or a limited group of shards, based on the shard key

For best performance, use targeted operations whenever possible. While some operations must broadcast to all shards, you can ensure MongoDB uses targeted operations whenever possible by always including the shard key.

Broadcast Operations mongos instances broadcast queries to all shards for the collection **unless** the mongos can determine which shard or subset of shards stores this data.

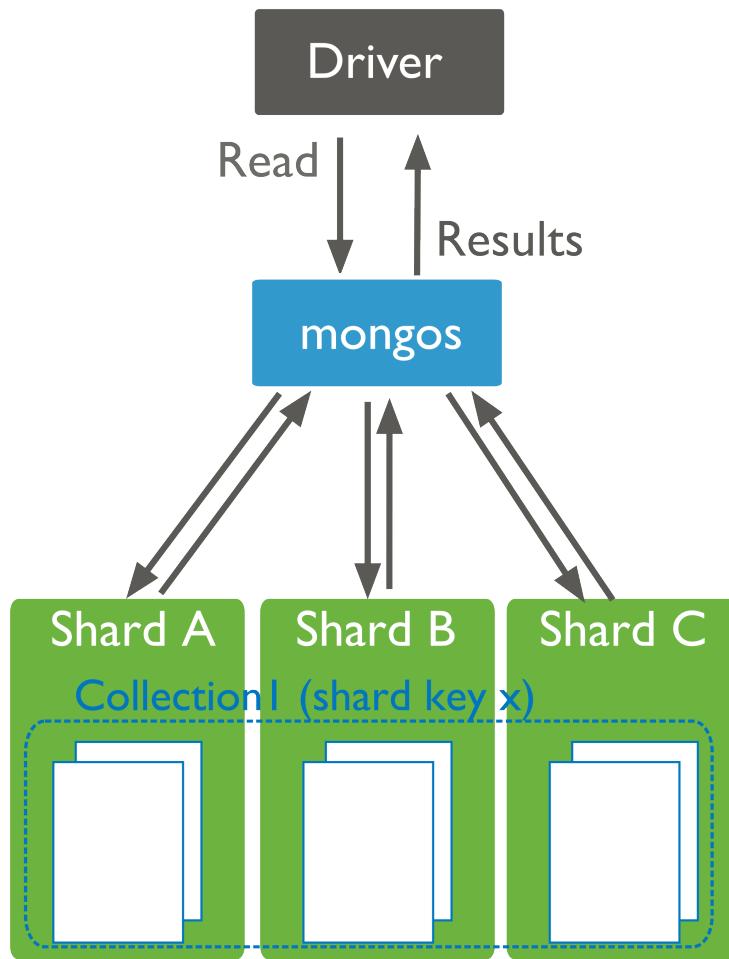


Figure 9.12: Read operations to a sharded cluster. Query criteria does not include the shard key. The query router mongos must broadcast query to all shards for the collection.

Multi-update operations are always broadcast operations.

The `remove()` operation is always a broadcast operation, unless the operation specifies the shard key in full.

Targeted Operations All `insert()` operations target to one shard.

All single `update()` (including `upsert` operations) and `remove()` operations must target to one shard.

Important: All single `update()` and `remove()` operations must include the *shard key or the `_id` field* in the query specification. `update()` or `remove()` operations that affect a single document in a sharded collection without the *shard key or the `_id` field* return an error.

For queries that include the shard key or portion of the shard key, `mongos` can target the query at a specific shard or set of shards. This is the case only if the portion of the shard key included in the query is a *prefix* of the shard key. For example, if the shard key is:

```
{ a: 1, b: 1, c: 1 }
```

The `mongos` program *can* route queries that include the full shard key or either of the following shard key prefixes at a specific shard or set of shards:

```
{ a: 1 }
{ a: 1, b: 1 }
```

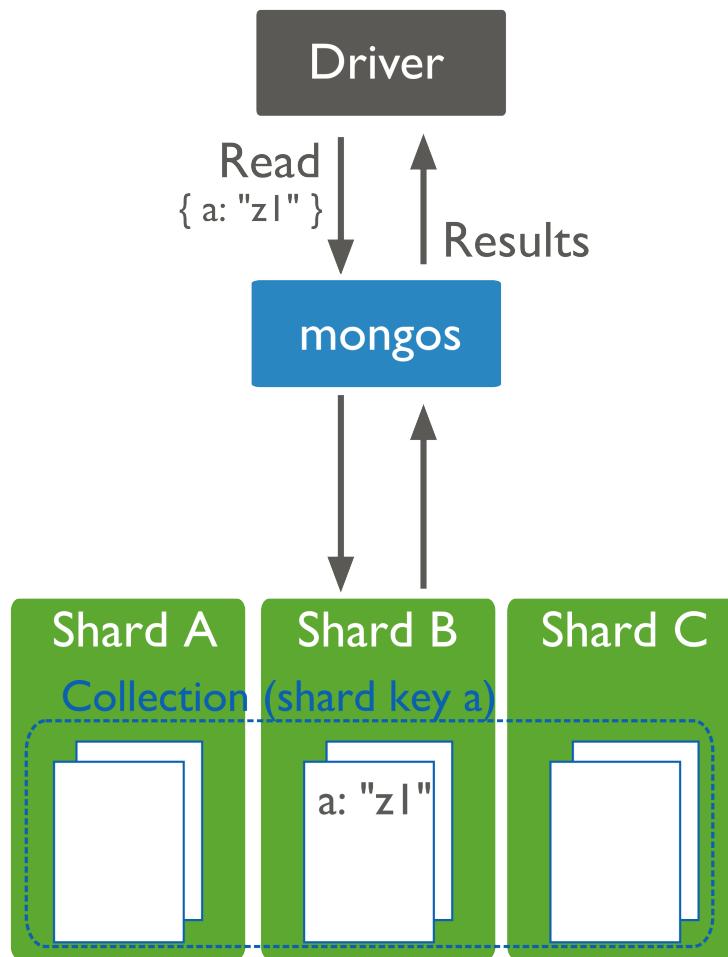


Figure 9.13: Read operations to a sharded cluster. Query criteria includes the shard key. The query router `mongos` can target the query to the appropriate shard or shards.

Depending on the distribution of data in the cluster and the selectivity of the query, `mongos` may still have to contact

multiple shards ⁵ to fulfill these queries.

Sharded and Non-Sharded Data

Sharding operates on the collection level. You can shard multiple collections within a database or have multiple databases with sharding enabled. ⁶ However, in production deployments, some databases and collections will use sharding, while other databases and collections will only reside on a single shard.

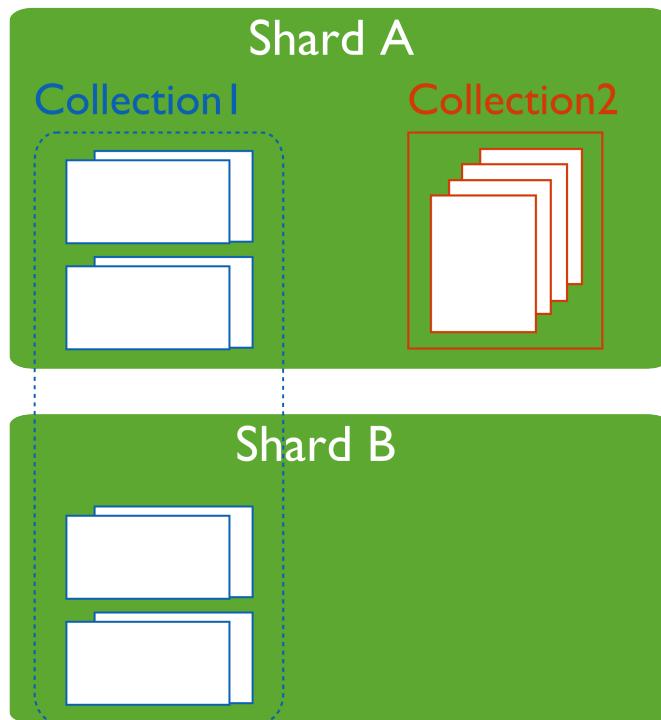


Figure 9.14: Diagram of a primary shard. A primary shard contains non-sharded collections as well as chunks of documents from sharded collections. Shard A is the primary shard.

Regardless of the data architecture of your *sharded cluster*, ensure that all queries and operations use the `mongos` router to access the data cluster. Use the `mongos` even for operations that do not impact the sharded data.

Related

[Sharded Cluster Security](#) (page 237)

9.2.4 Sharding Mechanics

The following documents describe sharded cluster processes.

[**Sharded Collection Balancing**](#) (page 508) Balancing distributes a sharded collection's data cluster to all of the shards.

⁵ `mongos` will route some queries, even some that include the shard key, to all shards, if needed.

⁶ As you configure sharding, you will use the `enableSharding` command to enable sharding for a database. This simply makes it possible to use the `shardCollection` command on a collection within that database.

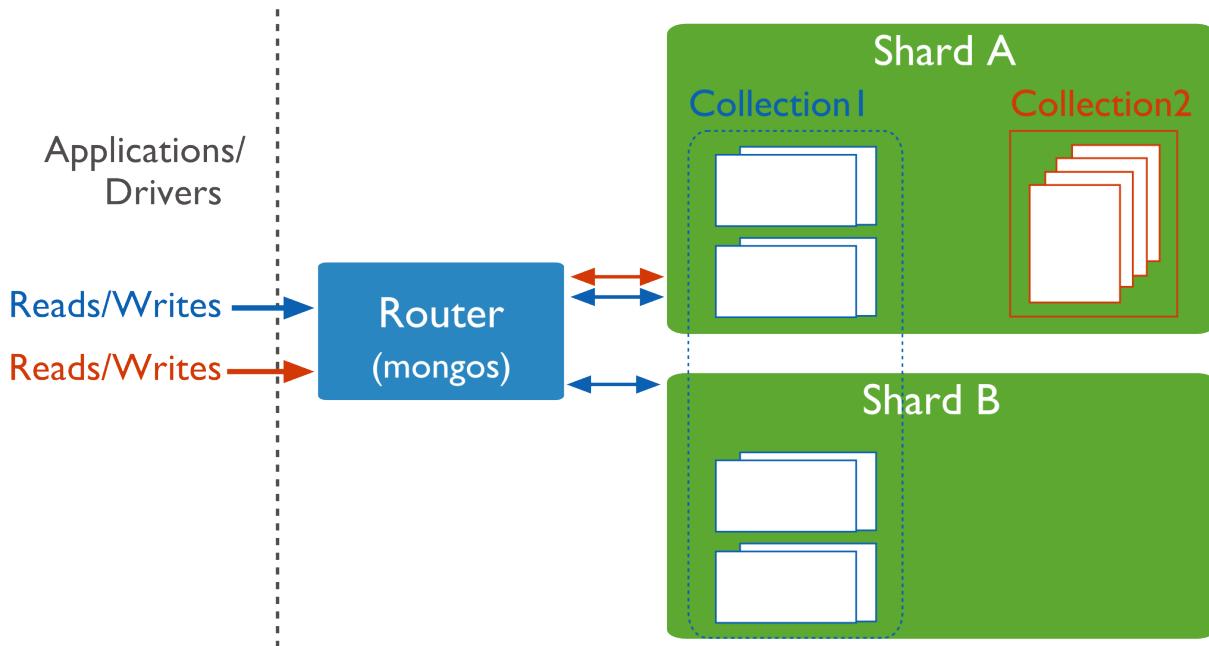


Figure 9.15: Diagram of applications/drivers issuing queries to mongos for unsharded collection as well as sharded collection. Config servers not shown.

Chunk Migration Across Shards (page 510) MongoDB migrates chunks to shards as part of the balancing process.

Chunk Splits in a Sharded Cluster (page 511) When a chunk grows beyond the configured size, MongoDB splits the chunk in half.

Shard Key Indexes (page 512) Sharded collections must keep an index that starts with the shard key.

Sharded Cluster Metadata (page 512) The cluster maintains internal metadata that reflects the location of data within the cluster.

Sharded Collection Balancing

Balancing is the process MongoDB uses to distribute data of a sharded collection evenly across a *sharded cluster*. When a *shard* has too many of a sharded collection's *chunks* compared to other shards, MongoDB automatically balances the the chunks across the shards. The balancing procedure for *sharded clusters* is entirely transparent to the user and application layer.

Cluster Balancer

The *balancer* process is responsible for redistributing the chunks of a sharded collection evenly among the shards for every sharded collection. By default, the balancer process is always running.

Any mongos instance in the cluster can start a balancing round. When a balancer process is active, the responsible mongos acquires a “lock” by modifying a document in the `lock` collection in the *Config Database* (page 555).

Note: Changed in version 2.0: Before MongoDB version 2.0, large differences in timekeeping (i.e. clock skew) between mongos instances could lead to failed distributed locks. This carries the possibility of data loss, particularly with skews larger than 5 minutes. Always use the network time protocol (NTP) by running ntpd on your servers to minimize clock skew.

To address uneven chunk distribution for a sharded collection, the balancer [migrates chunks](#) (page 510) from shards with more chunks to shards with a fewer number of chunks. The balancer migrates the chunks, one at a time, until there is an even dispersion of chunks for the collection across the shards.

Chunk migrations carry some overhead in terms of bandwidth and workload, both of which can impact database performance. The *balancer* attempts to minimize the impact by:

- Moving only one chunk at a time. See also [Chunk Migration Queuing](#) (page 511).
- Starting a balancing round **only** when the difference in the number of chunks between the shard with the greatest number of chunks for a sharded collection and the shard with the lowest number of chunks for that collection reaches the [migration threshold](#) (page 509).

You may disable the balancer temporarily for maintenance. See [Disable the Balancer](#) (page 540) for details.

You can also limit the window during which the balancer runs to prevent it from impacting production traffic. See [Schedule the Balancing Window](#) (page 539) for details.

Note: The specification of the balancing window is relative to the local time zone of all individual mongos instances in the cluster.

See also:

[Manage Sharded Cluster Balancer](#) (page 538).

Migration Thresholds

To minimize the impact of balancing on the cluster, the *balancer* will not begin balancing until the distribution of chunks for a sharded collection has reached certain thresholds. The thresholds apply to the difference in number of *chunks* between the shard with the most chunks for the collection and the shard with the fewest chunks for that collection. The balancer has the following thresholds:

Changed in version 2.2: The following thresholds appear first in 2.2. Prior to this release, a balancing round would only start if the shard with the most chunks had 8 more chunks than the shard with the least number of chunks.

Number of Chunks	Migration Threshold
Less than 20	2
21-80	4
Greater than 80	8

Once a balancing round starts, the balancer will not stop until, for the collection, the difference between the number of chunks on any two shards for that collection is *less than two* or a chunk migration fails.

Shard Size

By default, MongoDB will attempt to fill all available disk space with data on every shard as the data set grows. To ensure that the cluster always has the capacity to handle data growth, monitor disk usage as well as other performance metrics.

When adding a shard, you may set a “maximum size” for that shard. This prevents the *balancer* from migrating chunks to the shard when the value of `mapped` exceeds the “maximum size”. Use the `maxSize` parameter of the `addShard` command to set the “maximum size” for the shard.

See also:

[Change the Maximum Storage Size for a Given Shard](#) (page 537) and [Monitoring for MongoDB](#) (page 136).

Chunk Migration Across Shards

Chunk migration moves the chunks of a sharded collection from one shard to another and is part of the [balancer](#) (page 508) process.

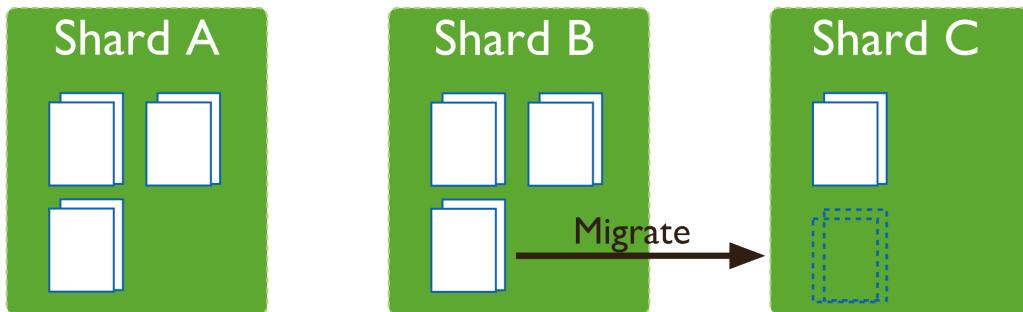


Figure 9.16: Diagram of a collection distributed across three shards. For this collection, the difference in the number of chunks between the shards reaches the [migration thresholds](#) (page 509) (in this case, 2) and triggers migration.

Chunk Migration

MongoDB migrates chunks in a *sharded cluster* to distribute the chunks of a sharded collection evenly among shards. Migrations may be either:

- Manual. Only use manual migration in limited cases, such as to distribute data during bulk inserts. See [Migrating Chunks Manually](#) (page 546) for more details.
- Automatic. The [balancer](#) (page 508) process automatically migrates chunks when there is an uneven distribution of a sharded collection's chunks across the shards. See [Migration Thresholds](#) (page 509) for more details.

All chunk migrations use the following procedure:

1. The balancer process sends the `moveChunk` command to the source shard.
2. The source starts the move with an internal `moveChunk` command. During the migration process, operations to the chunk route to the source shard. The source shard is responsible for incoming write operations for the chunk.
3. The destination shard begins requesting documents in the chunk and starts receiving copies of the data.
4. After receiving the final document in the chunk, the destination shard starts a synchronization process to ensure that it has the changes to the migrated documents that occurred during the migration.
5. When fully synchronized, the destination shard connects to the *config database* and updates the cluster metadata with the new location for the chunk.
6. After the destination shard completes the update of the metadata, and once there are no open cursors on the chunk, the source shard deletes its copy of the documents.

Changed in version 2.4: If the balancer needs to perform additional chunk migrations from the source shard, the balancer can start the next chunk migration without waiting for the current migration process to finish this deletion step. See [Chunk Migration Queuing](#) (page 511).

The migration process ensures consistency and maximizes the availability of chunks during balancing.

Chunk Migration Queuing

Changed in version 2.4.

To migrate multiple chunks from a shard, the balancer migrates the chunks one at a time. However, the balancer does not wait for the current migration's delete phase to complete before starting the next chunk migration. See [Chunk Migration](#) (page 510) for the chunk migration process and the delete phase.

This queuing behavior allows shards to unload chunks more quickly in cases of heavily imbalanced cluster, such as when performing initial data loads without pre-splitting and when adding new shards.

This behavior also affect the `moveChunk` command, and migration scripts that use the `moveChunk` command may proceed more quickly.

In some cases, the delete phases may persist longer. If multiple delete phases are queued but not yet complete, a crash of the replica set's primary can orphan data from multiple migrations.

Chunk Migration Write Concern Changed in version 2.4: While copying and deleting data during migrations, the balancer waits for [replication to secondaries](#) (page 47). This slows the potential speed of a chunk migration but increases reliability and ensures that a large number of chunk migrations *cannot* affect the availability of a sharded cluster.

See also [Secondary Throttle in the v2.2 Manual](#)⁷.

Chunk Splits in a Sharded Cluster

When a chunk grows beyond the [specified chunk size](#) (page 511), a `mongos` instance will split the chunk in half. Splits may lead to an uneven distribution of the chunks for a collection across the shards. In such cases, the `mongos` instances will initiate a round of migrations to redistribute chunks across shards. See [Sharded Collection Balancing](#) (page 508) for more details on balancing chunks across shards.

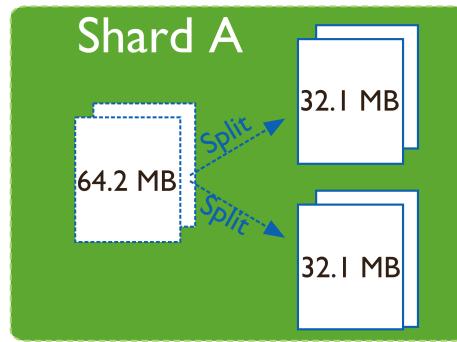


Figure 9.17: Diagram of a shard with a chunk that exceeds the default [chunk size](#) (page 511) of 64 MB and triggers a split of the chunk into two chunks.

Chunk Size

The default *chunk size* in MongoDB is 64 megabytes. You can [increase or reduce the chunk size](#) (page 547), mindful of its effect on the cluster's efficiency.

1. Small chunks lead to a more even distribution of data at the expense of more frequent migrations. This creates expense at the query routing (`mongos`) layer.

⁷<http://docs.mongodb.org/v2.2/tutorial/configure-sharded-cluster-balancer/#sharded-cluster-config-secondary-throttle>

2. Large chunks lead to fewer migrations. This is more efficient both from the networking perspective *and* in terms of internal overhead at the query routing layer. But, these efficiencies come at the expense of a potentially more uneven distribution of data.

For many deployments, it makes sense to avoid frequent and potentially spurious migrations at the expense of a slightly less evenly distributed data set.

Limitations

Changing the chunk size affects when chunks split but there are some limitations to its effects.

- Automatic splitting only occurs during inserts or updates. 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 must grow through inserts or updates until they reach the new size.

Note: Chunk ranges are inclusive of the lower boundary and exclusive of the upper boundary.

Shard Key Indexes

All sharded collections **must** have an index that starts with the *shard key*. If you shard a collection without any documents and *without* such an index, the `shardCollection` command will create the index on the shard key. If the collection already has documents, you must create the index before using `shardCollection`.

Changed in version 2.2: The index on the shard key no longer needs to be only on the shard key. This index can be an index of the shard key itself, or a *compound index* where the shard key is a prefix of the index.

Important: The index on the shard key **cannot** be a *multikey index* (page 320).

A sharded collection named `people` has for its shard key the field `zipcode`. It currently has the index `{ zipcode: 1 }`. You can replace this index with a compound index `{ zipcode: 1, username: 1 }`, as follows:

1. Create an index on `{ zipcode: 1, username: 1 }`:

```
db.people.ensureIndex( { zipcode: 1, username: 1 } );
```
2. When MongoDB finishes building the index, you can safely drop the existing index on `{ zipcode: 1 }`:

```
db.people.dropIndex( { zipcode: 1 } );
```

Since the index on the shard key cannot be a multikey index, the index `{ zipcode: 1, username: 1 }` can only replace the index `{ zipcode: 1 }` if there are no array values for the `username` field.

If you drop the last valid index for the shard key, recover by recreating an index on just the shard key.

For restrictions on shard key indexes, see *limits-shard-keys*.

Sharded Cluster Metadata

Config servers (page 496) store the metadata for a sharded cluster. The metadata reflects state and organization of the sharded data sets and system. The metadata includes the list of chunks on every shard and the ranges that define the chunks. The `mongos` instances cache this data and use it to route read and write operations to shards.

Config servers store the metadata in the *Config Database* (page 555).

Important: Always back up the `config` database before doing any maintenance on the config server.

To access the `config` database, issue the following command from the `mongo` shell:

```
use config
```

In general, you should *never* edit the content of the `config` database directly. The `config` database contains the following collections:

- [changelog](#) (page 556)
- [chunks](#) (page 557)
- [collections](#) (page 558)
- [databases](#) (page 558)
- [lockpings](#) (page 558)
- [locks](#) (page 558)
- [mongos](#) (page 559)
- [settings](#) (page 559)
- [shards](#) (page 560)
- [version](#) (page 560)

For more information on these collections and their role in sharded clusters, see [Config Database](#) (page 555). See [Read and Write Operations on Config Servers](#) (page 496) for more information about reads and updates to the metadata.

9.3 Sharded Cluster Tutorials

The following tutorials provide instructions for administering *sharded clusters*. For a higher-level overview, see [Sharding](#) (page 487).

Sharded Cluster Deployment Tutorials (page 514) Instructions for deploying sharded clusters, adding shards, selecting shard keys, and the initial configuration of sharded clusters.

Deploy a Sharded Cluster (page 514) 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 518) 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 520) 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 521) Add a shard to add capacity to a sharded cluster.

Continue reading from [Sharded Cluster Deployment Tutorials](#) (page 514) for additional tutorials.

Sharded Cluster Maintenance Tutorials (page 529) Procedures and tasks for common operations on active sharded clusters.

View Cluster Configuration (page 530) View status information about the cluster's databases, shards, and chunks.

Remove Shards from an Existing Sharded Cluster (page 541) Migrate a single shard's data and remove the shard.

[Migrate Config Servers with Different Hostnames](#) (page 532) 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 531) procedure.

[Manage Shard Tags](#) (page 548) Use tags to associate specific ranges of shard key values with specific shards.

Continue reading from [Sharded Cluster Maintenance Tutorials](#) (page 529) for additional tutorials.

[Sharded Cluster Data Management](#) (page 543) Practices that address common issues in managing large sharded data sets.

[Troubleshoot Sharded Clusters](#) (page 553) Presents solutions to common issues and concerns relevant to the administration and use of sharded clusters. Refer to [FAQ: MongoDB Diagnostics](#) (page 606) for general diagnostic information.

9.3.1 Sharded Cluster Deployment Tutorials

The following tutorials provide information on deploying sharded clusters.

[Deploy a Sharded Cluster](#) (page 514) 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 518) 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 520) Shard a collection based on hashes of a field's values in order to ensure even distribution over the collection's shards.

[Enable Authentication in a Sharded Cluster](#) (page 521) Control access to a sharded cluster through a key file and the keyFile setting on each of the cluster's components.

[Add Shards to a Cluster](#) (page 521) Add a shard to add capacity to a sharded cluster.

[Deploy Three Config Servers for Production Deployments](#) (page 522) 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 523) Convert a replica set to a sharded cluster in which each shard is its own replica set.

[Convert Sharded Cluster to Replica Set](#) (page 528) Replace your sharded cluster with a single replica set.

Deploy a Sharded Cluster

Deploy Sharded Cluster:

- [Start the Config Server Database Instances](#) (page 515)
- [Start the mongos Instances](#) (page 515)
- [Add Shards to the Cluster](#) (page 516)
- [Enable Sharding for a Database](#) (page 517)
- [Enable Sharding for a Collection](#) (page 517)

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 <http://docs.mongodb.org/manual/reference/program/mongod> or <http://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 fails and you receive the [Config Database String Error](#) (page 553) error.

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.

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" )
```

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: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> } )
```

Enable Sharding for a Collection

You enable sharding 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 519).
2. If the collection already contains data you must create an index on the *shard key* using `ensureIndex()`. If the collection is empty then MongoDB will create the index as part of the `sh.shardCollection()` step.
3. Enable sharding for 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 } )

db.alerts.ensureIndex( { _id : "hashed" } )
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 519) 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 519) 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 519) 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* (page 338), 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* (page 338) 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.”

See also:

[Cardinality](#) (page 519)

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:

[Write Scaling](#) (page 501)

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:

[Query Isolation](#) (page 501)

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](#) (page 501) or appropriate [write scaling](#) (page 501).

Shard a Collection Using a Hashed Shard Key

New in version 2.4.

[Hashed shard keys](#) (page 500) use a [hashed index](#) (page 338) 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 [Hashed Shard Keys](#) (page 500). For limitations on hashed indexes, see [Create a Hashed Index](#) (page 338).

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 consistently converted to 64-bit integers (and then back to floating point). MongoDB hashed indexes do not support floating point values larger than 2^{53} .

Enable Authentication in a Sharded Cluster

New in version 2.0: Support for authentication with sharded clusters.

To control access to a sharded cluster, create key files and then set the `keyFile` option on *all* components of the sharded cluster, including all `mongos` instances, all config server `mongod` instances, and all shard `mongod` instances. The content of the key file is arbitrary but must be the same on all cluster members.

Note: For an overview of authentication, see [Access Control](#) (page 235). For an overview of security, see [Security](#) (page 233).

Procedure

To enable authentication, do the following:

1. Generate a key file to store authentication information, as described in the [Generate a Key File](#) (page 256) section.
2. On each component in the sharded cluster, enable authentication by doing one of the following:
 - In the configuration file, set the `keyFile` option to the key file's path and then start the component, as in the following example:


```
keyFile = /srv/mongodb/keyfile
```
 - When starting the component, set `--keyFile` option, which is an option for both `mongos` instances and `mongod` instances. Set the `--keyFile` to the key file's path.

Note: The `keyFile` setting implies `auth`, which means in most cases you do not need to set `auth` explicitly.

3. Add the first administrative user and then add subsequent users. See [Create a User Administrator](#) (page 254).

Add Shards to a Cluster

You add shards to a *sharded cluster* after you create the cluster or anytime that you need to add capacity to the cluster. If you have not created a sharded cluster, see [Deploy a Sharded Cluster](#) (page 514).

When adding a shard to a cluster, you should always ensure that the cluster has enough capacity to support the migration without affecting legitimate production traffic.

In production environments, all shards should be *replica sets*.

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: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.

Deploy Three Config Servers for Production Deployments

This procedure converts a test deployment with only one [config server](#) (page 496) 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](#) (page 487) 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* (page 555), `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

Overview

Following this tutorial, you will convert a single 3-member replica set to a cluster that consists of 2 shards. Each shard will consist of an independent 3-member replica set.

The tutorial uses a test environment running on a local system UNIX-like system. You should feel encouraged to “follow along at home.” If you need to perform this process in a production environment, notes throughout the document indicate procedural differences.

The procedure, from a high level, is as follows:

1. Create or select a 3-member replica set and insert some data into a collection.
2. Start the config databases and create a cluster with a single shard.
3. Create a second replica set with three new `mongod` instances.
4. Add the second replica set as a shard in the cluster.
5. Enable sharding on the desired collection or collections.

Process

Install MongoDB according to the instructions in the *MongoDB Installation Tutorial* (page 3).

Deploy a Replica Set with Test Data If have an existing MongoDB *replica set* deployment, you can omit the this step and continue from *Deploy Sharding Infrastructure* (page 524).

Use the following sequence of steps to configure and deploy a replica set and to insert test data.

1. Create the following directories for the first replica set instance, named `firstset`:
 - `/data/example/firstset1`
 - `/data/example/firstset2`
 - `/data/example/firstset3`

To create directories, issue the following command:

```
mkdir -p /data/example/firstset1 /data/example/firstset2 /data/example/firstset3
```

2. In a separate terminal window or GNU Screen window, start three `mongod` instances by running each of the following commands:

```
mongod --dbpath /data/example/firstset1 --port 10001 --replSet firstset --oplogSize 700 --rest
mongod --dbpath /data/example/firstset2 --port 10002 --replSet firstset --oplogSize 700 --rest
mongod --dbpath /data/example/firstset3 --port 10003 --replSet firstset --oplogSize 700 --rest
```

Note: The `--oplogSize 700` option restricts the size of the operation log (i.e. oplog) for each mongod instance to 700MB. Without the `--oplogSize` option, each mongod reserves approximately 5% of the free disk space on the volume. By limiting the size of the oplog, each instance starts more quickly. Omit this setting in production environments.

3. In a mongo shell session in a new terminal, connect to the mongodb instance on port 10001 by running the following command. If you are in a production environment, first read the note below.

```
mongo localhost:10001/admin
```

Note: Above and hereafter, if you are running in a production environment or are testing this process with mongod instances on multiple systems, replace “localhost” with a resolvable domain, hostname, or the IP address of your system.

4. In the mongo shell, initialize the first replica set by issuing the following command:

```
db.runCommand({ "replSetInitiate" :
    { "_id" : "firstset", "members" : [ { "_id" : 1, "host" : "localhost:10001" },
        { "_id" : 2, "host" : "localhost:10002" },
        { "_id" : 3, "host" : "localhost:10003" }
    ] } )
{
    "info" : "Config now saved locally. Should come online in about a minute.",
    "ok" : 1
}
```

5. In the mongo shell, create and populate a new collection by issuing the following sequence of JavaScript operations:

```
use test
switched to db test
people = ["Marc", "Bill", "George", "Eliot", "Matt", "Trey", "Tracy", "Greg", "Steve", "Kristina"]
for(var i=0; i<1000000; i++) {
    name = people[Math.floor(Math.random()*people.length)];
    user_id = i;
    boolean = [true, false][Math.floor(Math.random()*2)];
    added_at = new Date();
    number = Math.floor(Math.random()*10001);
    db.test_collection.save({ "name":name, "user_id":user_id, "boolean":boolean });
}
```

The above operations add one million documents to the collection `test_collection`. This can take several minutes, depending on your system.

The script adds the documents in the following form:

```
{ "_id" : ObjectId("4ed5420b8fc1dd1df5886f70"), "name" : "Greg", "user_id" : 4, "boolean" : true, "a
```

Deploy Sharding Infrastructure This procedure creates the three config databases that store the cluster’s metadata.

Note: For development and testing environments, a single config database is sufficient. In production environments,

use three config databases. Because config instances store only the *metadata* for the sharded cluster, they have minimal resource requirements.

1. Create the following data directories for three *config database* instances:

- /data/example/config1
- /data/example/config2
- /data/example/config3

Issue the following command at the system prompt:

```
mkdir -p /data/example/config1 /data/example/config2 /data/example/config3
```

2. In a separate terminal window or GNU Screen window, start the config databases by running the following commands:

```
mongod --configsvr --dbpath /data/example/config1 --port 20001
mongod --configsvr --dbpath /data/example/config2 --port 20002
mongod --configsvr --dbpath /data/example/config3 --port 20003
```

3. In a separate terminal window or GNU Screen window, start mongos instance by running the following command:

```
mongos --configdb localhost:20001,localhost:20002,localhost:20003 --port 27017 --chunkSize 1
```

Note: If you are using the collection created earlier or are just experimenting with sharding, you can use a small `--chunkSize` (1MB works well.) The default `chunkSize` of 64MB means that your cluster must have 64MB of data before the MongoDB's automatic sharding begins working.

In production environments, do not use a small shard size.

The `configdb` options specify the *configuration databases* (e.g. `localhost:20001`, `localhost:20002`, and `localhost:2003`). The `mongos` instance runs on the default “MongoDB” port (i.e. 27017), while the databases themselves are running on ports in the 30001 series. In the this example, you may omit the `--port 27017` option, as 27017 is the default port.

4. Add the first shard in `mongos`. In a new terminal window or GNU Screen session, add the first shard, according to the following procedure:

- (a) Connect to the `mongos` with the following command:

```
mongo localhost:27017/admin
```

- (b) Add the first shard to the cluster by issuing the `addShard` command:

```
db.runCommand( { addShard : "firstset/localhost:10001,localhost:10002,localhost:10003" } )
```

- (c) Observe the following message, which denotes success:

```
{ "shardAdded" : "firstset", "ok" : 1 }
```

Deploy a Second Replica Set This procedure deploys a second replica set. This closely mirrors the process used to establish the first replica set above, omitting the test data.

1. Create the following data directories for the members of the second replica set, named `secondset`:

- /data/example/secondset1
- /data/example/secondset2

- /data/example/secondset3

2. In three new terminal windows, start three instances of mongod with the following commands:

```
mongod --dbpath /data/example/secondset1 --port 10004 --replSet secondset --oplogSize 700 --rest
mongod --dbpath /data/example/secondset2 --port 10005 --replSet secondset --oplogSize 700 --rest
mongod --dbpath /data/example/secondset3 --port 10006 --replSet secondset --oplogSize 700 --rest
```

Note: As above, the second replica set uses the smaller oplogSize configuration. Omit this setting in production environments.

3. In the mongo shell, connect to one mongodb instance by issuing the following command:

```
mongo localhost:10004/admin
```

4. In the mongo shell, initialize the second replica set by issuing the following command:

```
db.runCommand({ "replSetInitiate" :
    { "_id" : "secondset",
      "members" : [ { "_id" : 1, "host" : "localhost:10004" },
                    { "_id" : 2, "host" : "localhost:10005" },
                    { "_id" : 3, "host" : "localhost:10006" }
                ] } })
{
  "info" : "Config now saved locally. Should come online in about a minute.",
  "ok" : 1
}
```

5. Add the second replica set to the cluster. Connect to the mongos instance created in the previous procedure and issue the following sequence of commands:

```
use admin
db.runCommand( { addShard : "secondset/localhost:10004,localhost:10005,localhost:10006" } )
```

This command returns the following success message:

```
{ "shardAdded" : "secondset", "ok" : 1 }
```

6. Verify that both shards are properly configured by running the listShards command. View this and example output below:

```
db.runCommand({listShards:1})
{
  "shards" : [
    {
      "_id" : "firstset",
      "host" : "firstset/localhost:10001,localhost:10003,localhost:10002"
    },
    {
      "_id" : "secondset",
      "host" : "secondset/localhost:10004,localhost:10006,localhost:10005"
    }
  ],
  "ok" : 1
}
```

Enable Sharding MongoDB must have *sharding* enabled on *both* the database and collection levels.

Enabling Sharding on the Database Level Issue the `enableSharding` command. The following example enables sharding on the “`test`” database:

```
db.runCommand( { enableSharding : "test" } )
{ "ok" : 1 }
```

Create an Index on the Shard Key MongoDB uses the shard key to distribute documents between shards. Once selected, you cannot change the shard key. 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.

Typically shard keys are compound, comprising of some sort of hash and some sort of other primary key. Selecting a shard key depends on your data set, application architecture, and usage pattern, and is beyond the scope of this document. For the purposes of this example, we will shard the “`number`” key. This typically would *not* be a good shard key for production deployments.

Create the index with the following procedure:

```
use test
db.test_collection.ensureIndex({number:1})
```

See also:

The [Shard Key Overview](#) (page 499) and [Shard Key](#) (page 499) sections.

Shard the Collection Issue the following command:

```
use admin
db.runCommand( { shardCollection : "test.test_collection", key : {"number":1} })
{ "collectionssharded" : "test.test_collection", "ok" : 1 }
```

The collection `test_collection` is now sharded!

Over the next few minutes the Balancer begins to redistribute chunks of documents. You can confirm this activity by switching to the `test` database and running `db.stats()` or `db.printShardingStatus()`.

As clients insert additional documents into this collection, mongos distributes the documents evenly between the shards.

In the mongo shell, issue the following commands to return statics against each cluster:

```
use test
db.stats()
db.printShardingStatus()
```

Example output of the `db.stats()` command:

```
{
  "raw" : {
    "firstset/localhost:10001,localhost:10003,localhost:10002" : {
      "db" : "test",
      "collections" : 3,
      "objects" : 973887,
      "avgObjSize" : 100.33173458522396,
      "dataSize" : 97711772,
      "storageSize" : 141258752,
      "numExtents" : 15,
```

```
        "indexes" : 2,
        "indexSize" : 56978544,
        "fileSize" : 1006632960,
        "nsSizeMB" : 16,
        "ok" : 1
    },
    "secondset/localhost:10004,localhost:10006,localhost:10005" : {
        "db" : "test",
        "collections" : 3,
        "objects" : 26125,
        "avgObjSize" : 100.33286124401914,
        "dataSize" : 2621196,
        "storageSize" : 11194368,
        "numExtents" : 8,
        "indexes" : 2,
        "indexSize" : 2093056,
        "fileSize" : 201326592,
        "nsSizeMB" : 16,
        "ok" : 1
    }
},
"objects" : 1000012,
"avgObjSize" : 100.33176401883178,
"dataSize" : 100332968,
"storageSize" : 152453120,
"numExtents" : 23,
"indexes" : 4,
"indexSize" : 59071600,
"fileSize" : 1207959552,
"ok" : 1
}
```

Example output of the `db.printShardingStatus()` command:

```
--- Sharding Status ---
sharding version: { "_id" : 1, "version" : 3 }
shards:
    { "_id" : "firstset", "host" : "firstset/localhost:10001,localhost:10003,localhost:10002" }
    { "_id" : "secondset", "host" : "secondset/localhost:10004,localhost:10006,localhost:10005" }
databases:
    { "_id" : "admin", "partitioned" : false, "primary" : "config" }
    { "_id" : "test", "partitioned" : true, "primary" : "firstset" }
        test.test_collection chunks:
                                secondset      5
                                firstset     186
[...]
```

In a few moments you can run these commands for a second time to demonstrate that *chunks* are migrating from `firstset` to `secondset`.

When this procedure is complete, you will have converted a replica set into a cluster where each shard is itself a replica set.

Convert Sharded Cluster to Replica Set

- Convert a Cluster with a Single Shard into a Replica Set (page 529)
- Convert a Sharded Cluster into a Replica Set (page 529)

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 523). See the [Sharding](#) (page 487) 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 416) 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 185) 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.

9.3.2 Sharded Cluster Maintenance Tutorials

The following tutorials provide information in maintaining sharded clusters.

[View Cluster Configuration](#) (page 530) View status information about the cluster's databases, shards, and chunks.

[Migrate Config Servers with the Same Hostname \(page 531\)](#) 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 \(page 532\)](#) 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 \(page 531\)](#) procedure.

[Replace a Config Server \(page 533\)](#) Replaces a config server that has become inoperable. This procedure assumes that the hostname does not change.

[Migrate a Sharded Cluster to Different Hardware \(page 533\)](#) Migrate a sharded cluster to a different hardware system, for example, when moving a pre-production environment to production.

[Backup Cluster Metadata \(page 536\)](#) Create a backup of a sharded cluster's metadata while keeping the cluster operational.

[Configure Behavior of Balancer Process in Sharded Clusters \(page 537\)](#) Manage the balancer's behavior by scheduling a balancing window, changing size settings, or requiring replication before migration.

[Manage Sharded Cluster Balancer \(page 538\)](#) View balancer status and manage balancer behavior.

[Remove Shards from an Existing Sharded Cluster \(page 541\)](#) Migrate a single shard's data and remove the shard.

View Cluster Configuration

List Databases with Sharding Enabled

To list the databases that have sharding enabled, query the `databases` collection in the [Config Database \(page 555\)](#). 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" }
  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* (page 496) in a *sharded cluster* (page 492) 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

This procedure migrates a *config server* (page 496) in a *sharded cluster* (page 492) 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.

Changing a *config server's* (page 496) hostname **requires downtime** and requires restarting every process in the sharded cluster. 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 531).

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. Disable the cluster balancer process temporarily. See *Disable the Balancer* (page 540) for more information.
2. Shut down the config server.
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.

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 or *replica sets* that provide your shards.
- the `mongod` instances that provide your existing *config databases* (page 555).
- the `mongos` instances.

6. Restart all `mongod` processes that provide the shard servers.
7. Update the `configdb` setting for each `mongos` instances.
8. Restart the `mongos` instances.
9. Re-enable the balancer to allow the cluster to resume normal balancing operations. See the *Disable the Balancer* (page 540) section for more information on managing the balancer process.

Replace a Config Server

This procedure replaces an inoperable [config server](#) (page 496) in a [sharded cluster](#) (page 492). 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 532).

1. Disable the cluster balancer process temporarily. See [Disable the Balancer](#) (page 540) for more information.
2. Provision a new system, with the same hostname as the previous host.

You will have to ensure that 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.

3. Shut down *one* (and only one) of the existing config servers. Copy all of this host's dbpath file system tree 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
```

4. Restart the config server process that you used in the previous step to copy the data files to the new config server instance.
5. Start the new config server instance. The default invocation is:

```
mongod --configsvr
```

6. Re-enable the balancer to allow the cluster to resume normal balancing operations. See the [Disable the Balancer](#) (page 540) section for more information on managing the balancer process.

Note: In the course of this procedure *never* remove a config server from the configdb parameter on any of the mongos instances. If you need to change the name of a config server, always make sure that all mongos instances have three config servers specified in the configdb setting at all times.

Migrate a Sharded Cluster to Different Hardware

Migrate Sharded Cluster:

- [Disable the Balancer](#) (page 534)
- [Migrate Each Config Server Separately](#) (page 534)
- [Restart the mongos Instances](#) (page 534)
- [Migrate the Shards](#) (page 535)
 - [Migrate a Replica Set Shard](#) (page 535)
 - * [Migrate a Member of a Replica Set Shard](#) (page 535)
 - * [Migrate the Primary in a Replica Set Shard](#) (page 536)
 - [Migrate a Standalone Shard](#) (page 536)
- [Re-Enable the Balancer](#) (page 536)

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](#) (page 512). 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 428) 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* (page 510) 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 540).

Migrate Each Config Server Separately

Migrate each *config server* (page 496) 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 532) 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 515).

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](#) (page 385) 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](#) (page 474) 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 [Example Reconfiguration Operations](#) (page 477).

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](#) (page 392) 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](#) (page 481) state. To migrate the stepped-down primary, follow the [Migrate a Member of a Replica Set Shard](#) (page 535) 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 428) and then use the procedure for [migrating a replica set shard](#) (page 535). 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 541) procedure.

Re-Enable the Balancer

To complete the migration, re-enable the balancer to resume [chunk migrations](#) (page 510).

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 541).

Backup Cluster Metadata

This procedure shuts down the `mongod` instance of a [config server](#) (page 496) in order to create a backup of a [sharded cluster’s](#) (page 487) 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 operational⁸.

1. Disable the cluster balancer process temporarily. See [Disable the Balancer](#) (page 540) 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 540) section for more information on managing the balancer process.

⁸ 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 [Config Servers](#) (page 496) for more information.

See also:

[Backup Strategies for MongoDB Systems](#) (page 134).

Configure Behavior of Balancer Process in Sharded Clusters

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 [Sharded Collection Balancing](#) (page 508) and [Cluster Balancer](#) (page 508).

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 539)
- [Remove a Balancing Window Schedule](#) (page 540).

The `mongos` instances user their own local timezones to 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 [Chunk Size](#) (page 511).

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 547).

Change the Maximum Storage Size for a Given Shard

The `maxSize` field in the `shards` (page 560) collection in the `config database` (page 555) sets the maximum size for a shard, allowing you to control whether the balancer will migrate chunks to a shard. If mapped size⁹ 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.

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:

⁹ This value includes the mapped size of all data files including the “local” and `admin` databases. Account for this when setting `maxSize`.

```
db.runCommand( { addshard : "example.net:34008", maxSize : 125 } )
```

To set `maxSize` on an existing shard, insert or update the `maxSize` field in the `shards` (page 560) collection in the `config database` (page 555). 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 } } )
```

Require Replication before Chunk Migration (Secondary Throttle)

New in version 2.2.1: `_secondaryThrottle` became an option to the balancer and to command `moveChunk`. `_secondaryThrottle` makes it possible to require the balancer wait for replication to secondaries during migrations.

Changed in version 2.4: `_secondaryThrottle` became the default mode for all balancer and `moveChunk` operations.

Before 2.2.1, the write operations required to migrate chunks between shards do not need to replicate to secondaries in order to succeed. However, you can configure the balancer to require migration related write operations to replicate to secondaries. This throttles or slows the migration process and in doing so reduces the potential impact of migrations on a sharded cluster.

You can throttle migrations by enabling the balancer's `_secondaryThrottle` parameter. When enabled, secondary throttle requires a `{ w : 2 }` write concern on delete and insertion operations, so that every operation propagates to at least one secondary before the balancer issues the next operation.

Starting with version 2.4 the default `secondaryThrottle` value is `true`. To revert to previous behavior, set `_secondaryThrottle` to `false`.

You enable or disable `_secondaryThrottle` directly in the `settings` (page 559) collection in the `config database` (page 555) by running the following commands from a mongo shell, connected to a mongos instance:

```
use config
db.settings.update( { "_id" : "balancer" } , { $set : { "_secondaryThrottle" : true } } , { upsert : true } )
```

You also can enable secondary throttle when issuing the `moveChunk` command by setting `_secondaryThrottle` to `true`. For more information, see `moveChunk`.

Manage Sharded Cluster Balancer

This page describes common administrative procedures related to balancing. For an introduction to balancing, see `Sharded Collection Balancing` (page 508). For lower level information on balancing, see `Cluster Balancer` (page 508).

See also:

[Configure Behavior of Balancer Process in Sharded Clusters](#) (page 537)

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()
```

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](#) (page 555):

```
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](#) (page 555):

```
use config
```

3. Use an operation modeled on the following example `update()` operation to modify the balancer's window:

```
db.settings.update({ _id : "balancer" }, { $set : { activeWindow : { start : "<start-time>", stop : "<end-time>" } } })
```

Replace `<start-time>` and `<end-time>` with time values using two digit hour and minute values (e.g. `HH:MM`) that describe the beginning and end boundaries of the balancing window. These 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 : "06:00" } } })
```

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.

Remove a Balancing Window Schedule

If you have [set the balancing window](#) (page 539) 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.setBalancerState(false)
```

If a migration is in progress, the system will complete the in-progress migration before stopping.

3. To verify that the balancer has stopped, issue the following command, which returns `false` if the balancer is stopped:

```
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.startBalancer()` helper, issue the following command from the `config` database:

```
db.settings.update( { _id: "balancer" }, { $set : { stopped: true } } , 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 } } , true )
```

Disable Balancing During Backups

If MongoDB migrates a *chunk* during a *backup* (page 134), 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 539) 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 540) 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.

Remove Shards from an Existing Sharded Cluster

Remove Shards:

- Ensure the Balancer Process is Enabled (page 542)
- Determine the Name of the Shard to Remove (page 542)
- Remove Chunks from the Shard (page 542)
- Check the Status of the Migration (page 542)
- Move Unsharded Data (page 543)
- Finalize the Migration (page 543)

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 540).

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

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:

```
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`. For example, for a shard named `mongodb0`, run:

```
db.runCommand( { removeShard: "mongodb0" } )
```

The command returns output similar to the following:

```
{ msg: "draining ongoing" , state: "ongoing" , remaining: { chunks: NumberLong(42) , dbs : NumberLong
```

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. Then proceed to the next step.

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 543).

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:

```
db.runCommand( { removeShard: "mongodb0" } )
```

A success message appears at completion:

```
{ msg: "remove shard completed successfully", state: "completed", host: "mongodb0", ok : 1 }
```

Once the value of the `stage` field is “completed”, you may safely stop the processes comprising the `mongodb0` shard.

See also:

[Backup and Restore Sharded Clusters](#) (page 193)

9.3.3 Sharded Cluster Data Management

The following documents provide information in managing data in sharded clusters.

[Create Chunks in a Sharded Cluster](#) (page 544) Create chunks, or *pre-split* empty collection to ensure an even distribution of chunks during data ingestion.

[Split Chunks in a Sharded Cluster](#) (page 545) Manually create chunks in a sharded collection.

[Migrate Chunks in a Sharded Cluster \(page 546\)](#) Manually migrate chunks without using the automatic balance process.

[Modify Chunk Size in a Sharded Cluster \(page 547\)](#) Modify the default chunk size in a sharded collection

[Tag Aware Sharding \(page 547\)](#) Tags associate specific ranges of *shard key* values with specific shards for use in managing deployment patterns.

[Manage Shard Tags \(page 548\)](#) Use tags to associate specific ranges of shard key values with specific shards.

[Enforce Unique Keys for Sharded Collections \(page 550\)](#) Ensure that a field is always unique in all collections in a sharded cluster.

[Shard GridFS Data Store \(page 552\)](#) 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 [Cluster Balancer](#) (page 508) and [Chunk Migration](#) (page 510). For information on manually migrating chunks, see [Migrate Chunks in a Sharded Cluster](#) (page 546).

Split Chunks in a Sharded Cluster

Normally, MongoDB splits a *chunk* after an insert if the chunk exceeds the maximum [chunk size](#) (page 511). 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: Chunks cannot be merged or combined once they've been split.

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 539), 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 *Cluster Balancer* (page 508) and *Chunk Migration* (page 510).

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.ex
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 544) 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 *Require Replication before Chunk Migration (Secondary Throttle)* (page 538).

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.

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.

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](#) (page 555):

```
use config
```

3. Issue the following `save()` operation to store the global chunk size configuration value:

```
db.settings.save( { _id:"chunkszie", value: <size> } )
```

Note: The `chunkSize` and `--chunkSize` options, passed at runtime 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 runtime 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.

Tag Aware Sharding

MongoDB supports tagging a range of *shard key* values to associate that range with a shard or group of shards. Those shards receive all inserts within the tagged range.

The balancer obeys tagged range associations, which enables the following deployment patterns:

- isolate a specific subset of data on a specific set of shards.
- ensure that the most relevant data reside on shards that are geographically closest to the application servers.

This document describes the behavior, operation, and use of tag aware sharding in MongoDB deployments.

Note: *Shard key* range tags are distinct from [replica set member tags](#) (page 403).

Important: *Hash-based sharding* does not support tag-aware sharding.

Behavior and Operations

The balancer migrates chunks of documents in a sharded collections to the shards associated with a tag that has a *shard key* range with an *upper bound greater* than the chunk's *lower bound*.

Note: If the chunks in sharded collection are already balanced, then the balancer will not migrate any chunks. If chunks in a sharded collection are not balanced, the balancer migrates chunks in tagged ranges to shards associated with those tags.

After configuring tags with a shard key range, and associating it with a shard or shards, the cluster may take some time to balance the data among the shards. This depends on the division of chunks and the current distribution of data in the cluster.

Once configured, the balancer respects tag ranges during future *balancing rounds* (page 508).

See also:

[Manage Shard Tags](#) (page 548)

Chunks that Span Multiple Tag Ranges

A single chunk may contain data with a *shard key* values that falls into ranges associated with more than one tag. To accommodate these situations, the balancer may migrate chunks to shards that contain shard key values that exceed the upper bound of the selected tag range.

Example

Given a sharded collection with two configured tag ranges:

- *Shard key* values between 100 and 200 have tags to direct corresponding chunks to shards tagged NYC.
- *Shard key* values between 200 and 300 have tags to direct corresponding chunks to shards tagged SFO.

For this collection cluster, the balancer will migrate a chunk with *shard key* values ranging between 150 and 220 to a shard tagged NYC, since 150 is closer to 200 than 300.

To ensure that your collection has no potentially ambiguously tagged chunks, [create splits on your tag boundaries](#) (page 545). You can then manually migrate chunks to the appropriate shards, or wait for the balancer to automatically migrate these chunks.

Manage Shard Tags

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` (page 560) collection of the `config` database.

Each document in the `tags` (page 560) 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` (page 560) 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` (page 560) 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" })
```

Enforce Unique Keys for Sharded Collections

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.¹⁰

If your need to ensure that a field is always unique in all collections in a sharded environment, there are two options:

1. Enforce uniqueness of the *shard key* (page 499).

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* (page 329).

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.

Note: 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.

Always use the default `acknowledged` (page 45) `write concern` (page 44) in conjunction with a *recent MongoDB driver* (page 654).

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" } )
```

¹⁰ If you specify a unique index on a sharded collection, MongoDB will be able to enforce uniqueness only among the documents located on a single shard *at the time of creation*.

Warning: In any sharded collection where you are *not* sharding by the `_id` field, you must ensure uniqueness of the `_id` field. The best way to ensure `_id` is always unique is to use `ObjectId`, or another universally unique identifier (UUID.)

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* (page 501) and *query isolation* (page 501), as well as *high cardinality* (page 519). These ideal shard keys are not often the same keys that require uniqueness and 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 550); 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.ensureIndex( { "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*:

```
use 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: `ensureIndex()` 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.

Shard GridFS Data Store

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.ensureIndex( { 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 : { file_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.

9.3.4 Troubleshoot Sharded Clusters

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.

9.4 Sharding Reference

9.4.1 Sharding Methods in the mongo Shell

Name	Description
sh._adminCommand	Runs a <i>database command</i> against the admin database, like db.runCommand(), but can confirm that it is issued against a mongos.
sh._checkFullName	Tests a namespace to determine if its well formed.
sh._checkMongos()	Tests to see if the mongo shell is connected to a mongos instance.
sh._lastMigration	Reports on the last <i>chunk</i> migration.
sh.addShard()	Adds a <i>shard</i> to a sharded cluster.
sh.addShardTag()	Associates a shard with a tag, to support <i>tag aware sharding</i> (page 547).
sh.addTagRange()	Associates range of shard keys with a shard tag, to support <i>tag aware sharding</i> (page 547).
sh.disableBalancing()	Disable balancing on a single collection in a sharded database. Does not affect balancing of other collections in a sharded cluster.
sh.enableBalancing()	Activates the sharded collection balancer process if previously disabled using sh.disableBalancing().
sh.enableSharding()	Enables sharding on a specific database.
sh.getBalancerHost()	Returns the name of a mongos that's responsible for the balancer process.
sh.getBalancerStatus()	Returns a boolean to report if the <i>balancer</i> is currently enabled.
sh.help()	Returns help text for the sh methods.
sh.isBalancerRunning()	Returns a boolean to report if the balancer process is currently migrating chunks.
sh.moveChunk()	Migrates a <i>chunk</i> in a <i>sharded cluster</i> .
sh.removeShardTag()	Removes the association between a shard and a shard tag shard tag.
sh.setBalancerState()	Enables or disables the <i>balancer</i> which migrates <i>chunks</i> between <i>shards</i> .
sh.shardCollection()	Enables sharding for a collection.
sh.splitAt()	Divides an existing <i>chunk</i> into two chunks using a specific value of the <i>shard key</i> as the dividing point.
sh.splitFind()	Divides an existing <i>chunk</i> that contains a document matching a query into two approximately equal chunks.
sh.startBalancer()	Enables the <i>balancer</i> and waits for balancing to start.
sh.status()	Reports on the status of a <i>sharded cluster</i> , as db.printShardingStatus().
sh.stopBalancer()	Disables the <i>balancer</i> and waits for any in progress balancing rounds to complete.
sh.waitForBalance()	Internal. Waits for the balancer state to change.
sh.waitForBalance()	Internal. Waits until the balancer stops running.
sh.waitForDLock()	Internal. Waits for a specified distributed <i>sharded cluster</i> lock.
sh.waitForPingChange()	Internal. Waits for a change in ping state from one of the mongos in the sharded cluster.

9.4.2 Sharding Database Commands

The following database commands support *sharded clusters*.

Name	Description
flushRouterConfig	Forces an update to the cluster metadata cached by a mongos.
addShard	Adds a <i>shard</i> to a <i>sharded cluster</i> .
checkShardingIndex	Internal command that validates index on shard key.
enableSharding	Enables sharding on a specific database.
listShards	Returns a list of configured shards.
removeShard	Starts the process of removing a shard from a sharded cluster.
getShardMap	Internal command that reports on the state of a sharded cluster.
getShardVersion	Internal command that returns the <i>config server</i> version.
setShardVersion	Internal command to sets the <i>config server</i> version.
shardCollection	Enables the sharding functionality for a collection, allowing the collection to be sharded.
shardingState	Reports whether the mongod is a member of a sharded cluster.
unsetSharding	Internal command that affects connections between instances in a MongoDB deployment.
split	Creates a new <i>chunk</i> .
splitChunk	Internal command to split chunk. Instead use the methods <code>sh.splitFind()</code> and <code>sh.splitAt()</code> .
splitVector	Internal command that determines split points.
medianKey	Deprecated internal command. See <code>splitVector</code> .
moveChunk	Internal command that migrates chunks between shards.
movePrimary	Reassigns the <i>primary shard</i> when removing a shard from a sharded cluster.
isdbgrid	Verifies that a process is a mongos.

9.4.3 Reference Documentation

Config Database (page 555) Complete documentation of the content of the `local` database that MongoDB uses to store sharded cluster metadata.

Sharding Command Quick Reference (page 561) A quick reference for all *commands* and `mongo` shell methods that support sharding and sharded clusters.

Config Database

The config database supports *sharded cluster* operation. See the *Sharding* (page 487) section of this manual for full documentation of sharded clusters.

Important: Consider the schema of the config database *internal* and may change between releases of MongoDB. The config database is not a dependable API, and users should not write data to the config database in the course of normal operation or maintenance.

Warning: Modification of the config database on a functioning system may lead to instability or inconsistent data sets. If you must modify the config database, use `mongodump` to create a full backup of the config database.

To access the config database, connect to a mongos instance in a sharded cluster, and use the following helper:

```
use config
```

You can return a list of the collections, with the following helper:

```
show collections
```

Collections

config

config.changelog

Internal MongoDB Metadata

The config (page 556) database is internal: applications and administrators should not modify or depend upon its content in the course of normal operation.

The changelog (page 556) collection stores a document for each change to the metadata of a sharded collection.

Example

The following example displays a single record of a chunk split from a changelog (page 556) collection:

```
{  
    "_id" : "<hostname>-<timestamp>-<increment>",  
    "server" : "<hostname><:port>",  
    "clientAddr" : "127.0.0.1:63381",  
    "time" : ISODate("2012-12-11T14:09:21.039Z"),  
    "what" : "split",  
    "ns" : "<database>.<collection>",  
    "details" : {  
        "before" : {  
            "min" : {  
                "<database>" : { $minKey : 1 }  
            },  
            "max" : {  
                "<database>" : { $maxKey : 1 }  
            },  
            "lastmod" : Timestamp(1000, 0),  
            "lastmodEpoch" : ObjectId("00000000000000000000000000000000")  
        },  
        "left" : {  
            "min" : {  
                "<database>" : { $minKey : 1 }  
            },  
            "max" : {  
                "<database>" : "<value>"  
            },  
            "lastmod" : Timestamp(1000, 1),  
            "lastmodEpoch" : ObjectId(<...>)  
        },  
        "right" : {  
            "min" : {  
                "<database>" : "<value>"  
            },  
            "max" : {  
                "<database>" : { $maxKey : 1 }  
            },  
            "lastmod" : Timestamp(1000, 2),  
            "lastmodEpoch" : ObjectId("<...>")  
        }  
    }  
}
```

Each document in the [changelog](#) (page 556) collection contains the following fields:

config.changelog._id

The value of `changelog._id` is: <hostname>-<timestamp>-<increment>.

config.changelog.server

The hostname of the server that holds this data.

config.changelog.clientAddr

A string that holds the address of the client, a mongos instance that initiates this change.

config.changelog.time

A `ISODate` timestamp that reflects when the change occurred.

config.changelog.what

Reflects the type of change recorded. Possible values are:

- `dropCollection`
- `dropCollection.start`
- `dropDatabase`
- `dropDatabase.start`
- `moveChunk.start`
- `moveChunk.commit`
- `split`
- `multi-split`

config.changelog.ns

Namespace where the change occurred.

config.changelog.details

A document that contains additional details regarding the change. The structure of the [details](#) (page 557) document depends on the type of change.

config.chunks

Internal MongoDB Metadata

The [config](#) (page 556) database is internal: applications and administrators should not modify or depend upon its content in the course of normal operation.

The [chunks](#) (page 557) collection stores a document for each chunk in the cluster. Consider the following example of a document for a chunk named `records.pets-animal_\"cat\"`:

```
{
  "_id" : "mydb.foo-a_\"cat\"",
  "lastmod" : Timestamp(1000, 3),
  "lastmodEpoch" : ObjectId("5078407bd58b175c5c225fdc"),
  "ns" : "mydb.foo",
  "min" : {
    "animal" : "cat"
  },
  "max" : {
    "animal" : "dog"
  },
}
```

```
        "shard" : "shard0004"
    }
```

These documents store the range of values for the shard key that describe the chunk in the `min` and `max` fields. Additionally the `shard` field identifies the shard in the cluster that “owns” the chunk.

config.collections

Internal MongoDB Metadata

The `config` (page 556) database is internal: applications and administrators should not modify or depend upon its content in the course of normal operation.

The `collections` (page 558) collection stores a document for each sharded collection in the cluster. Given a collection named `pets` in the `records` database, a document in the `collections` (page 558) collection would resemble the following:

```
{
  "_id" : "records.pets",
  "lastmod" : ISODate("1970-01-16T15:00:58.107Z"),
  "dropped" : false,
  "key" : {
    "a" : 1
  },
  "unique" : false,
  "lastmodEpoch" : ObjectId("5078407bd58b175c5c225fdc")
}
```

config.databases

Internal MongoDB Metadata

The `config` (page 556) database is internal: applications and administrators should not modify or depend upon its content in the course of normal operation.

The `databases` (page 558) collection stores a document for each database in the cluster, and tracks if the database has sharding enabled. `databases` (page 558) represents each database in a distinct document. When a databases have sharding enabled, the `primary` field holds the name of the *primary shard*.

```
{ "_id" : "admin", "partitioned" : false, "primary" : "config" }
{ "_id" : "mydb", "partitioned" : true, "primary" : "shard0000" }
```

config.lockpings

Internal MongoDB Metadata

The `config` (page 556) database is internal: applications and administrators should not modify or depend upon its content in the course of normal operation.

The `lockpings` (page 558) collection keeps track of the active components in the sharded cluster. Given a cluster with a `mongos` running on `example.com:30000`, the document in the `lockpings` (page 558) collection would resemble:

```
{ "_id" : "example.com:30000:1350047994:16807", "ping" : ISODate("2012-10-12T18:32:54.892Z") }
```

config.locks**Internal MongoDB Metadata**

The [config](#) (page 556) database is internal: applications and administrators should not modify or depend upon its content in the course of normal operation.

The [locks](#) (page 558) collection stores a distributed lock. This ensures that only one `mongos` instance can perform administrative tasks on the cluster at once. The `mongos` acting as *balancer* takes a lock by inserting a document resembling the following into the `locks` collection.

```
{
    "_id" : "balancer",
    "process" : "example.net:40000:1350402818:16807",
    "state" : 2,
    "ts" : ObjectId("507daeedf40e1879df62e5f3"),
    "when" : ISODate("2012-10-16T19:01:01.593Z"),
    "who" : "example.net:40000:1350402818:16807:Balancer:282475249",
    "why" : "doing balance round"
}
```

If a `mongos` holds the balancer lock, the `state` field has a value of 2, which means that balancer is active. The `when` field indicates when the balancer began the current operation.

Changed in version 2.0: The value of the `state` field was 1 before MongoDB 2.0.

config.mongos**Internal MongoDB Metadata**

The [config](#) (page 556) database is internal: applications and administrators should not modify or depend upon its content in the course of normal operation.

The [mongos](#) (page 559) collection stores a document for each `mongos` instance affiliated with the cluster. `mongos` instances send pings to all members of the cluster every 30 seconds so the cluster can verify that the `mongos` is active. The `ping` field shows the time of the last ping, while the `up` field reports the uptime of the `mongos` as of the last ping. The cluster maintains this collection for reporting purposes.

The following document shows the status of the `mongos` running on `example.com:30000`.

```
{ "_id" : "example.com:30000", "ping" : ISODate("2012-10-12T17:08:13.538Z"), "up" : 13699, "wait"
```

config.settings**Internal MongoDB Metadata**

The [config](#) (page 556) database is internal: applications and administrators should not modify or depend upon its content in the course of normal operation.

The [settings](#) (page 559) collection holds the following sharding configuration settings:

- Chunk size. To change chunk size, see [Modify Chunk Size in a Sharded Cluster](#) (page 547).
- Balancer status. To change status, see [Disable the Balancer](#) (page 540).

The following is an example `settings` collection:

```
{ "_id" : "chunksize", "value" : 64 }
{ "_id" : "balancer", "stopped" : false }

config.shards
```

Internal MongoDB Metadata

The [config](#) (page 556) database is internal: applications and administrators should not modify or depend upon its content in the course of normal operation.

The [shards](#) (page 560) collection represents each shard in the cluster in a separate document. If the shard is a replica set, the `host` field displays the name of the replica set, then a slash, then the hostname, as in the following example:

```
{ "_id" : "shard0000", "host" : "shard1/localhost:30000" }
```

If the shard has [tags](#) (page 547) assigned, this document has a `tags` field, that holds an array of the tags, as in the following example:

```
{ "_id" : "shard0001", "host" : "localhost:30001", "tags": [ "NYC" ] }
```

```
config.tags
```

Internal MongoDB Metadata

The [config](#) (page 556) database is internal: applications and administrators should not modify or depend upon its content in the course of normal operation.

The [tags](#) (page 560) collection holds documents for each tagged shard key range in the cluster. The documents in the [tags](#) (page 560) collection resemble the following:

```
{
  "_id" : { "ns" : "records.users", "min" : { "zipcode" : "10001" } },
  "ns" : "records.users",
  "min" : { "zipcode" : "10001" },
  "max" : { "zipcode" : "10281" },
  "tag" : "NYC"
}
```

```
config.version
```

Internal MongoDB Metadata

The [config](#) (page 556) database is internal: applications and administrators should not modify or depend upon its content in the course of normal operation.

The [version](#) (page 560) collection holds the current metadata version number. This collection contains only one document:

To access the [version](#) (page 560) collection you must use the `db.getCollection()` method. For example, to display the collection's document:

```
mongos> db.getCollection("version").find()
{ "_id" : 1, "version" : 3 }
```

Note: Like all databases in MongoDB, the `config` database contains a `system.indexes` (page 223) collection which contains metadata for all indexes in the database for information on indexes, see [Indexes](#) (page 309).

Sharding Command Quick Reference

JavaScript Methods

Definition

`sh.addShard(host)`

Adds a database instance or replica set to a *sharded cluster*. The optimal configuration is to deploy shards across *replica sets*. This method must be run on a `mongos` instance.

The `sh.addShard()` method has the following parameter:

param string host The hostname of either a standalone database instance or of a replica set. Include the port number if the instance is running on a non-standard port. Include the replica set name if the instance is a replica set, as explained below.

The `sh.addShard()` method has the following prototype form:

```
sh.addShard("<host>")
```

The `host` parameter can be in any of the following forms:

- [hostname]
- [hostname] : [port]
- [replica-set-name] / [hostname]
- [replica-set-name] / [hostname] : port

Warning: Do not use `localhost` for the hostname unless your *configuration server* is also running on `localhost`.

The `sh.addShard()` method is a helper for the `addShard` command. The `addShard` command has additional options which are not available with this helper.

Important: You cannot include a *hidden member* (page 383) in the seed list provided to `addShard`.

Example

To add a shard on a replica set, specify the name of the replica set and the hostname of at least one member of the replica set, as a seed. If you specify additional hostnames, all must be members of the same replica set.

The following example adds a replica set named `rep10` and specifies one member of the replica set:

```
sh.addShard("rep10/mongodb3.example.net:27327")
```

Definition

`sh.enableSharding(database)`

Enables sharding on the specified database. This does not automatically shard any collections but makes it possible to begin sharding collections using `sh.shardCollection()`.

The `sh.enableSharding()` method has the following parameter:

param string database The name of the database shard. Enclose the name in quotation marks.

See also:

`sh.shardCollection()`

Definition

`sh.shardCollection(namespace, key, unique)`

Shards a collection using the `key` as a the *shard key*. `sh.shardCollection()` takes the following arguments:

param string namespace The *namespace* of the collection to shard.

param document key A *document* that specifies the *shard key* to use to *partition* and distribute objects among the shards. A shard key may be one field or multiple fields. A shard key with multiple fields is called a “compound shard key.”

param Boolean unique When true, ensures that the underlying index enforces a unique constraint. *Hashed shard keys* do not support unique constraints.

New in version 2.4: Use the form `{field: "hashed"}` to create a *hashed shard key*. Hashed shard keys may not be compound indexes.

Warning: MongoDB provides no method to deactivate sharding for a collection after calling `shardCollection`. Additionally, after `shardCollection`, you cannot change shard keys or modify the value of any field used in your shard key index.

See also:

`shardCollection` for additional options, [Sharding](#) (page 487) and [Sharding Introduction](#) (page 487) for an overview of sharding, [Deploy a Sharded Cluster](#) (page 514) for a tutorial, and [Shard Keys](#) (page 499) for choosing a shard key.

Example

Given the `people` collection in the `records` database, the following command shards the collection by the `zipcode` field:

```
sh.shardCollection("records.people", { zipcode: 1 })
```

Definition

`sh.splitFind(namespace, query)`

Splits the chunk containing the document specified by the `query` at its median point, creating two roughly equal chunks. Use `sh.splitAt()` to split a collection in a specific point.

In most circumstances, you should leave chunk splitting to the automated processes. However, when initially deploying a *sharded cluster* it is necessary to perform some measure of *pre-splitting* using manual methods including `sh.splitFind()`.

param string namespace The namespace (i.e. `<database>.<collection>`) of the sharded collection that contains the chunk to split.

param document query A query to identify a document in a specific chunk. Typically specify the *shard key* for a document as the query.

Definition

`sh.splitAt(namespace, query)`

Splits the chunk containing the document specified by the query as if that document were at the “middle” of the collection, even if the specified document is not the actual median of the collection.

param string namespace The *namespace* (i.e. <database>.<collection>) of the sharded collection that contains the chunk to split.

param document query A query to identify a document in a specific chunk. Typically specify the *shard key* for a document as the query.

Use this command to manually split chunks unevenly. Use the “`sh.splitFind()`” function to split a chunk at the actual median.

In most circumstances, you should leave chunk splitting to the automated processes within MongoDB. However, when initially deploying a *sharded cluster* it is necessary to perform some measure of *pre-splitting* using manual methods including `sh.splitAt()`.

Definition

`sh.moveChunk(namespace, query, destination)`

Moves the *chunk* that contains the document specified by the *query* to the *destination shard*. `sh.moveChunk()` provides a wrapper around the `moveChunk` database command and takes the following arguments:

param string namespace The *namespace* of the sharded collection that contains the chunk to migrate.

param document query An equality match on the shard key that selects the chunk to move.

param string destination The name of the shard to move.

Important: In most circumstances, allow the *balancer* to automatically migrate *chunks*, and avoid calling `sh.moveChunk()` directly.

See also:

`moveChunk`, `sh.splitAt()`, `sh.splitFind()`, [Sharding](#) (page 487), and [chunk migration](#) (page 510).

Example

Given the `people` collection in the `records` database, the following operation finds the chunk that contains the documents with the `zipcode` field set to `53187` and then moves that chunk to the shard named `shard0019`:

```
sh.moveChunk("records.people", { zipcode: 53187 }, "shard0019")
```

Description

`sh.setBalancerState(state)`

Enables or disables the *balancer*. Use `sh.getBalancerState()` to determine if the balancer is currently enabled or disabled and `sh.isBalancerRunning()` to check its current state.

The `sh.getBalancerState()` method has the following parameter:

param Boolean state Set this to `true` to enable the balancer and `false` to disable it.

See also:

- `sh.enableBalancing()`
- `sh.disableBalancing()`
- `sh.getBalancerHost()`
- `sh.getBalancerState()`
- `sh.isBalancerRunning()`
- `sh.startBalancer()`
- `sh.stopBalancer()`
- `sh.waitForBalancer()`
- `sh.waitForBalancerOff()`

`sh.isBalancerRunning()`

Returns boolean

Returns true if the *balancer* process is currently running and migrating chunks and false if the balancer process is not running. Use `sh.getBalancerState()` to determine if the balancer is enabled or disabled.

See also:

- `sh.enableBalancing()`
- `sh.disableBalancing()`
- `sh.getBalancerHost()`
- `sh.getBalancerState()`
- `sh.setBalancerState()`
- `sh.startBalancer()`
- `sh.stopBalancer()`
- `sh.waitForBalancer()`
- `sh.waitForBalancerOff()`

`sh.status()`

Prints a formatted report of the sharding configuration and the information regarding existing chunks in a *sharded cluster*. The default behavior suppresses the detailed chunk information if the total number of chunks is greater than or equal to 20.

The `sh.status()` method has the following parameter:

param Boolean verbose If `true`, the method displays details of the document distribution across chunks when you have 20 or more chunks.

See also:

`db.printShardingStatus()`

Definition

`sh.addShardTag(shard, tag)`

New in version 2.2.

Associates a shard with a tag or identifier. MongoDB uses these identifiers to direct *chunks* that fall within a tagged range to specific shards. `sh.addTagRange()` associates chunk ranges with tag ranges.

param string shard The name of the shard to which to give a specific tag.

param string tag The name of the tag to add to the shard.

Only issue `sh.addShardTag()` when connected to a `mongos` instance.

Example

The following example adds three tags, NYC, LAX, and NRT, to three shards:

```
sh.addShardTag("shard0000", "NYC")
sh.addShardTag("shard0001", "LAX")
sh.addShardTag("shard0002", "NRT")
```

See also:

`sh.addTagRange()` and `sh.removeShardTag()`.

Definition

`sh.addTagRange(namespace, minimum, maximum, tag)`

New in version 2.2.

Attaches a range of shard key values to a shard tag created using the `sh.addShardTag()` method. `sh.addTagRange()` takes the following arguments:

param string namespace The *namespace* of the sharded collection to tag.

param document minimum The minimum value of the *shard key* range to include in the tag. Specify the minimum value in the form of `<fieldname>:<value>`. This value must be of the same BSON type or types as the shard key.

param document maximum The maximum value of the shard key range to include in the tag. Specify the maximum value in the form of `<fieldname>:<value>`. This value must be of the same BSON type or types as the shard key.

param string tag The name of the tag to attach the range specified by the `minimum` and `maximum` arguments to.

Use `sh.addShardTag()` to ensure that the balancer migrates documents that exist within the specified range to a specific shard or set of shards.

Only issue `sh.addTagRange()` when connected to a `mongos` instance.

Note: If you add a tag range to a collection using `sh.addTagRange()` and then later drop the collection or its database, MongoDB does not remove the tag association. If you later create a new collection with the same name, the old tag association will apply to the new collection.

Example

Given a shard key of { state: 1, zip: 1 }, the following operation creates a tag range covering zip codes in New York State:

```
sh.addTagRange( "exampledbs.collection",
    { state: "NY", zip: MinKey },
    { state: "NY", zip: MaxKey },
    "NY"
)
```

Definition

`sh.removeShardTag(shard, tag)`

New in version 2.2.

Removes the association between a tag and a shard. Only issue `sh.removeShardTag()` when connected to a `mongos` instance.

param string shard The name of the shard from which to remove a tag.

param string tag The name of the tag to remove from the shard.

See also:

`sh.addShardTag()`, `sh.addTagRange()`

`sh.help()`

Returns a basic help text for all sharding related shell functions.

Database Commands

The following database commands support *sharded clusters*.

Definition

`addShard`

Adds either a database instance or a *replica set* to a *sharded cluster*. The optimal configuration is to deploy shards across replica sets.

Run `addShard` when connected to a `mongos` instance. The command takes the following form when adding a single database instance as a shard:

```
{ addShard: "<hostname><:port>", maxSize: <size>, name: "<shard_name>" }
```

When adding a replica set as a shard, use the following form:

```
{ addShard: "<replica_set>/<hostname><:port>", maxSize: <size>, name: "<shard_name>" }
```

The command contains the following fields:

field string addShard The hostname and port of the `mongod` instance to be added as a shard. To add a replica set as a shard, specify the name of the replica set and the hostname and port of a member of the replica set.

field integer maxSize The maximum size in megabytes of the shard. If you set `maxSize` to 0, MongoDB does not limit the size of the shard.

field string name A name for the shard. If this is not specified, MongoDB automatically provides a unique name.

The addShard command stores shard configuration information in the *config database*.

Specify a maxSize when you have machines with different disk capacities, or if you want to limit the amount of data on some shards. The maxSize constraint prevents the *balancer* from migrating chunks to the shard when the value of mem.mapped exceeds the value of maxSize.

Important: You cannot include a *hidden member* (page 383) in the seed list provided to addShard.

Examples

The following command adds the database instance running on port “27027“ on the host `mongodb0.example.net` as a shard:

```
db.runCommand({addShard: "mongodb0.example.net:27027"})
```

Warning: Do not use `localhost` for the hostname unless your *configuration server* is also running on `localhost`.

The following command adds a replica set as a shard:

```
db.runCommand( { addShard: "rep10/mongodb3.example.net:27327" } )
```

You may specify all members in the replica set. All additional hostnames must be members of the same replica set.

listShards

Use the `listShards` command to return a list of configured shards. The command takes the following form:

```
{ listShards: 1 }
```

enableSharding

The `enableSharding` command enables sharding on a per-database level. Use the following command form:

```
{ enableSharding: "<database name>" }
```

Once you’ve enabled sharding in a database, you can use the `shardCollection` command to begin the process of distributing data among the shards.

Definition

shardCollection

Enables a collection for sharding and allows MongoDB to begin distributing data among shards. You must run `enableSharding` on a database before running the `shardCollection` command. `shardCollection` has the following form:

```
{ shardCollection: "<database>.<collection>", key: <shardkey> }
```

`shardCollection` has the following fields:

field string shardCollection The *namespace* of the collection to shard in the form `<database>.<collection>`.

field document key The index specification document to use as the shard key. The index must exist prior to the `shardCollection` command, unless the collection is empty. If the collection is empty, in which case MongoDB creates the index prior to sharding the collection. New in

version 2.4: The key may be in the form { field : "hashed" }, which will use the specified field as a hashed shard key.

field Boolean unique When true, the unique option ensures that the underlying index enforces a unique constraint. Hashed shard keys do not support unique constraints.

field integer numInitialChunks To support *hashed sharding* (page 500) added in MongoDB 2.4, numInitialChunks specifies the number of chunks to create when sharding an collection with a hashed shard key. MongoDB will then create and balance chunks across the cluster. The numInitialChunks must be less than 8192.

Warning: Do not run more than one shardCollection command on the same collection at the same time.

Shard Keys Choosing the best shard key to effectively distribute load among your shards requires some planning. Review *Shard Keys* (page 499) regarding choosing a shard key.

Hashed Shard Keys New in version 2.4.

Hashed shard keys (page 500) use a hashed index of a single field as the shard key.

Warning: MongoDB provides no method to deactivate sharding for a collection after calling shardCollection. Additionally, after shardCollection, you cannot change shard keys or modify the value of any field used in your shard key index.

See also:

Sharding (page 487), *Sharding Concepts* (page 492), and *Deploy a Sharded Cluster* (page 514).

Example

The following operation enables sharding for the people collection in the records database and uses the zipcode field as the *shard key* (page 499):

```
db.runCommand( { shardCollection: "records.people", key: { zipcode: 1 } } )
```

shardingState

shardingState is an admin command that reports if mongod is a member of a *sharded cluster*. shardingState has the following prototype form:

```
{ shardingState: 1 }
```

For shardingState to detect that a mongod is a member of a sharded cluster, the mongod must satisfy the following conditions:

- 1.the mongod is a primary member of a replica set, and
- 2.the mongod instance is a member of a sharded cluster.

If shardingState detects that a mongod is a member of a sharded cluster, shardingState returns a document that resembles the following prototype:

```
{
  "enabled" : true,
  "configServer" : "<configdb-string>",
  "shardName" : "<string>",
  "shardHost" : "string:",
```

```

"versions" : {
    "<database>.<collection>" : Timestamp(<...>),
    "<database>.<collection>" : Timestamp(<...>)
},
"ok" : 1
}

```

Otherwise, shardingState will return the following document:

```
{ "note" : "from execCommand", "ok" : 0, "errmsg" : "not master" }
```

The response from shardingState when used with a *config server* is:

```
{ "enabled": false, "ok": 1 }
```

Note: mongos instances do not provide the shardingState.

Warning: This command obtains a write lock on the affected database and will block other operations until it has completed; however, the operation is typically short lived.

removeShard

Starts the process of removing a shard from a *cluster*. This is a multi-stage process. Begin by issuing the following command:

```
{ removeShard : "[shardName]" }
```

The balancer will then migrate chunks from the shard specified by [shardName]. This process happens slowly to avoid placing undue load on the overall cluster.

The command returns immediately, with the following message:

```
{ msg : "draining started successfully" , state: "started" , shard: "shardName" , ok : 1 }
```

If you run the command again, you'll see the following progress output:

```
{ msg: "draining ongoing" , state: "ongoing" , remaining: { chunks: 23 , dbs: 1 } , ok: 1 }
```

The remaining *document* specifies how many chunks and databases remain on the shard. Use db.printShardingStatus() to list the databases that you must move from the shard.

Each database in a sharded cluster has a primary shard. If the shard you want to remove is also the primary of one of the cluster's databases, then you must manually move the database to a new shard. This can be only after the shard is empty. See the movePrimary command for details.

After removing all chunks and databases from the shard, you may issue the command again, to return:

```
{ msg: "remove shard completed successfully" , state: "completed" , host: "shardName" , ok : 1 }
```

Frequently Asked Questions

10.1 FAQ: MongoDB Fundamentals

Frequently Asked Questions:

- What kind of database is MongoDB? (page 571)
- Do MongoDB databases have tables? (page 572)
- Do MongoDB databases have schemas? (page 572)
- What languages can I use to work with MongoDB? (page 572)
- Does MongoDB support SQL? (page 572)
- What are typical uses for MongoDB? (page 572)
- Does MongoDB support transactions? (page 573)
- Does MongoDB require a lot of RAM? (page 573)
- How do I configure the cache size? (page 573)
- Does MongoDB require a separate caching layer for application-level caching? (page 573)
- Does MongoDB handle caching? (page 574)
- Are writes written to disk immediately, or lazily? (page 574)
- What language is MongoDB written in? (page 574)
- What are the limitations of 32-bit versions of MongoDB? (page 574)

This document addresses basic high level questions about MongoDB and its use.

If you don't find the answer you're looking for, check the [complete list of FAQs](#) (page 571) or post your question to the [MongoDB User Mailing List](#)¹.

10.1.1 What kind of database is MongoDB?

MongoDB is a *document-oriented* DBMS. Think of MySQL but with *JSON*-like objects comprising the data model, rather than RDBMS tables. Significantly, MongoDB supports neither joins nor transactions. However, it features secondary indexes, an expressive query language, atomic writes on a per-document level, and fully-consistent reads.

Operationally, MongoDB features master-slave replication with automated failover and built-in horizontal scaling via automated range-based partitioning.

Note: MongoDB uses *BSON*, a binary object format similar to, but more expressive than *JSON*.

¹<https://groups.google.com/forum/?fromgroups#!forum/mongodb-user>

10.1.2 Do MongoDB databases have tables?

Instead of tables, a MongoDB database stores its data in *collections*, which are the rough equivalent of RDBMS tables. A collection holds one or more *documents*, which corresponds to a record or a row in a relational database table, and each document has one or more fields, which corresponds to a column in a relational database table.

Collections have important differences from RDBMS tables. Documents in a single collection may have a unique combination and set of fields. Documents need not have identical fields. You can add a field to some documents in a collection without adding that field to all documents in the collection.

See

SQL to MongoDB Mapping Chart (page 82)

10.1.3 Do MongoDB databases have schemas?

MongoDB uses dynamic schemas. You can create collections without defining the structure, i.e. the fields or the types of their values, of the documents in the collection. You can change the structure of documents simply by adding new fields or deleting existing ones. Documents in a collection need not have an identical set of fields.

In practice, it is common for the documents in a collection to have a largely homogeneous structure; however, this is not a requirement. MongoDB's flexible schemas mean that schema migration and augmentation are very easy in practice, and you will rarely, if ever, need to write scripts that perform "alter table" type operations, which simplifies and facilitates iterative software development with MongoDB.

See

SQL to MongoDB Mapping Chart (page 82)

10.1.4 What languages can I use to work with MongoDB?

MongoDB *client drivers* exist for all of the most popular programming languages, and many other ones. See the [latest list of drivers](#)² for details.

See also:

MongoDB Drivers and Client Libraries (page 92).

10.1.5 Does MongoDB support SQL?

No.

However, MongoDB does support a rich, ad-hoc query language of its own.

See also:

<http://docs.mongodb.org/manual/reference/operator>

10.1.6 What are typical uses for MongoDB?

MongoDB has a general-purpose design, making it appropriate for a large number of use cases. Examples include content management systems, mobile applications, gaming, e-commerce, analytics, archiving, and logging.

²<http://docs.mongodb.org/ecosystem/drivers>

Do not use MongoDB for systems that require SQL, joins, and multi-object transactions.

10.1.7 Does MongoDB support transactions?

MongoDB does not provide ACID transactions.

However, MongoDB does provide some basic transactional capabilities. Atomic operations are possible within the scope of a single document: that is, we can debit `a` and credit `b` as a transaction if they are fields within the same document. Because documents can be rich, some documents contain thousands of fields, with support for testing fields in sub-documents.

Additionally, you can make writes in MongoDB durable (the ‘D’ in ACID). To get durable writes, you must enable journaling, which is on by default in 64-bit builds. You must also issue writes with a write concern of `{ j: true }` to ensure that the writes block until the journal has synced to disk.

Users have built successful e-commerce systems using MongoDB, but applications requiring multi-object commits with rollback generally aren’t feasible.

10.1.8 Does MongoDB require a lot of RAM?

Not necessarily. It’s certainly possible to run MongoDB on a machine with a small amount of free RAM.

MongoDB automatically uses all free memory on the machine as its cache. System resource monitors show that MongoDB uses a lot of memory, but its usage is dynamic. If another process suddenly needs half the server’s RAM, MongoDB will yield cached memory to the other process.

Technically, the operating system’s virtual memory subsystem manages MongoDB’s memory. This means that MongoDB will use as much free memory as it can, swapping to disk as needed. Deployments with enough memory to fit the application’s working data set in RAM will achieve the best performance.

See also:

FAQ: MongoDB Diagnostics (page 606) for answers to additional questions about MongoDB and Memory use.

10.1.9 How do I configure the cache size?

MongoDB has no configurable cache. MongoDB uses all *free* memory on the system automatically by way of memory-mapped files. Operating systems use the same approach with their file system caches.

10.1.10 Does MongoDB require a separate caching layer for application-level caching?

No. In MongoDB, a document’s representation in the database is similar to its representation in application memory. This means the database already stores the usable form of data, making the data usable in both the persistent store and in the application cache. This eliminates the need for a separate caching layer in the application.

This differs from relational databases, where caching data is more expensive. Relational databases must transform data into object representations that applications can read and must store the transformed data in a separate cache: if these transformation from data to application objects require joins, this process increases the overhead related to using the database which increases the importance of the caching layer.

10.1.11 Does MongoDB handle caching?

Yes. MongoDB keeps all of the most recently used data in RAM. If you have created indexes for your queries and your working data set fits in RAM, MongoDB serves all queries from memory.

MongoDB does not implement a query cache: MongoDB serves all queries directly from the indexes and/or data files.

10.1.12 Are writes written to disk immediately, or lazily?

Writes are physically written to the *journal* (page 229) within 100 milliseconds, by default. At that point, the write is “durable” in the sense that after a pull-plug-from-wall event, the data will still be recoverable after a hard restart. See `journalCommitInterval` for more information on the journal commit window.

While the journal commit is nearly instant, MongoDB writes to the data files lazily. MongoDB may wait to write data to the data files for as much as one minute by default. This does not affect durability, as the journal has enough information to ensure crash recovery. To change the interval for writing to the data files, see `syncdelay`.

10.1.13 What language is MongoDB written in?

MongoDB is implemented in C++. *Drivers* and client libraries are typically written in their respective languages, although some drivers use C extensions for better performance.

10.1.14 What are the limitations of 32-bit versions of MongoDB?

MongoDB uses *memory-mapped files* (page 600). When running a 32-bit build of MongoDB, the total storage size for the server, including data and indexes, is 2 gigabytes. For this reason, do not deploy MongoDB to production on 32-bit machines.

If you’re running a 64-bit build of MongoDB, there’s virtually no limit to storage size. For production deployments, 64-bit builds and operating systems are strongly recommended.

See also:

“Blog Post: 32-bit Limitations³“

Note: 32-bit builds disable *journaling* by default because journaling further limits the maximum amount of data that the database can store.

10.2 FAQ: MongoDB for Application Developers

³<http://blog.mongodb.org/post/137788967/32-bit-limitations>

Frequently Asked Questions:

- What is a namespace in MongoDB? (page 575)
- How do you copy all objects from one collection to another? (page 575)
- If you remove a document, does MongoDB remove it from disk? (page 576)
- When does MongoDB write updates to disk? (page 576)
- How do I do transactions and locking in MongoDB? (page 576)
- How do you aggregate data with MongoDB? (page 576)
- Why does MongoDB log so many “Connection Accepted” events? (page 577)
- Does MongoDB run on Amazon EBS? (page 577)
- Why are MongoDB’s data files so large? (page 577)
- How do I optimize storage use for small documents? (page 577)
- When should I use GridFS? (page 578)
- How does MongoDB address SQL or Query injection? (page 578)
 - BSON (page 578)
 - JavaScript (page 579)
 - Dollar Sign Operator Escaping (page 579)
 - Driver-Specific Issues (page 580)
- How does MongoDB provide concurrency? (page 580)
- What is the compare order for BSON types? (page 580)
- How do I query for fields that have null values? (page 581)
- Are there any restrictions on the names of Collections? (page 582)
- How do I isolate cursors from intervening write operations? (page 582)
- When should I embed documents within other documents? (page 583)
- Where can I learn more about data modeling in MongoDB? (page 583)
- Can I manually pad documents to prevent moves during updates? (page 583)

This document answers common questions about application development using MongoDB.

If you don’t find the answer you’re looking for, check the [complete list of FAQs](#) (page 571) or post your question to the MongoDB User Mailing List⁴.

10.2.1 What is a namespace in MongoDB?

A “namespace” is the concatenation of the *database* name and the *collection* names⁵ with a period character in between.

Collections are containers for documents that share one or more indexes. Databases are groups of collections stored on disk using a single set of data files.⁶

For an example `acme.users` namespace, `acme` is the database name and `users` is the collection name. Period characters **can** occur in collection names, so that `acme.user.history` is a valid namespace, with `acme` as the database name, and `user.history` as the collection name.

While data models like this appear to support nested collections, the collection namespace is flat, and there is no difference from the perspective of MongoDB between `acme`, `acme.users`, and `acme.records`.

10.2.2 How do you copy all objects from one collection to another?

In the `mongo` shell, you can use the following operation to duplicate the entire collection:

⁴<https://groups.google.com/forum/?fromgroups#!forum/mongodb-user>

⁵ Each index also has its own namespace.

⁶ MongoDB database have a configurable limit on the number of namespaces in a database.

```
db.source.copyTo(newCollection)
```

Warning: When using `db.collection.copyTo()` check field types to ensure that the operation does not remove type information from documents during the translation from *BSON* to *JSON*. Consider using `cloneCollection()` to maintain type fidelity.

Also consider the `cloneCollection` command that may provide some of this functionality.

10.2.3 If you remove a document, does MongoDB remove it from disk?

Yes.

When you use `remove()`, the object will no longer exist in MongoDB's on-disk data storage.

10.2.4 When does MongoDB write updates to disk?

MongoDB flushes writes to disk on a regular interval. In the default configuration, MongoDB writes data to the main data files on disk every 60 seconds and commits the *journal* roughly every 100 milliseconds. These values are configurable with the `journalCommitInterval` and `syncdelay`.

These values represent the *maximum* amount of time between the completion of a write operation and the point when the write is durable in the journal, if enabled, and when MongoDB flushes data to the disk. In many cases MongoDB and the operating system flush data to disk more frequently, so that the above values represents a theoretical maximum.

However, by default, MongoDB uses a "lazy" strategy to write to disk. This is advantageous in situations where the database receives a thousand increments to an object within one second, MongoDB only needs to flush this data to disk once. In addition to the aforementioned configuration options, you can also use `fSync` and `getLastError` to modify this strategy.

10.2.5 How do I do transactions and locking in MongoDB?

MongoDB does not have support for traditional locking or complex transactions with rollback. MongoDB aims to be lightweight, fast, and predictable in its performance. This is similar to the MySQL MyISAM autocommit model. By keeping transaction support extremely simple, MongoDB can provide greater performance especially for *partitioned* or *replicated* systems with a number of database server processes.

MongoDB *does* have support for atomic operations *within* a single document. Given the possibilities provided by nested documents, this feature provides support for a large number of use-cases.

See also:

The [Isolate Sequence of Operations](#) (page 74) page.

10.2.6 How do you aggregate data with MongoDB?

In version 2.1 and later, you can use the new *aggregation framework* (page 277), with the `aggregate` command.

MongoDB also supports *map-reduce* with the `mapReduce` command, as well as basic aggregation with the `group`, `count`, and `distinct`. commands.

See also:

The [Aggregation](#) (page 273) page.

10.2.7 Why does MongoDB log so many “Connection Accepted” events?

If you see a very large number connection and re-connection messages in your MongoDB log, then clients are frequently connecting and disconnecting to the MongoDB server. This is normal behavior for applications that do not use request pooling, such as CGI. Consider using FastCGI, an Apache Module, or some other kind of persistent application server to decrease the connection overhead.

If these connections do not impact your performance you can use the run-time `quiet` option or the command-line option `--quiet` to suppress these messages from the log.

10.2.8 Does MongoDB run on Amazon EBS?

Yes.

MongoDB users of all sizes have had a great deal of success using MongoDB on the EC2 platform using EBS disks.

See also:

[Amazon EC2](#)⁷

10.2.9 Why are MongoDB’s data files so large?

MongoDB aggressively preallocates data files to reserve space and avoid file system fragmentation. You can use the `smallfiles` setting to modify the file preallocation strategy.

See also:

Why are the files in my data directory larger than the data in my database? (page 601)

10.2.10 How do I optimize storage use for small documents?

Each MongoDB document contains a certain amount of overhead. This overhead is normally insignificant but becomes significant if all documents are just a few bytes, as might be the case if the documents in your collection only have one or two fields.

Consider the following suggestions and strategies for optimizing storage utilization for these collections:

- Use the `_id` field explicitly.

MongoDB clients automatically add an `_id` field to each document and generate a unique 12-byte *ObjectId* for the `_id` field. Furthermore, MongoDB always indexes the `_id` field. For smaller documents this may account for a significant amount of space.

To optimize storage use, users can specify a value for the `_id` field explicitly when inserting documents into the collection. This strategy allows applications to store a value in the `_id` field that would have occupied space in another portion of the document.

You can store any value in the `_id` field, but because this value serves as a primary key for documents in the collection, it must uniquely identify them. If the field’s value is not unique, then it cannot serve as a primary key as there would be collisions in the collection.

- Use shorter field names.

MongoDB stores all field names in every document. For most documents, this represents a small fraction of the space used by a document; however, for small documents the field names may represent a proportionally large amount of space. Consider a collection of documents that resemble the following:

⁷<http://docs.mongodb.org/ecosystem/platforms/amazon-ec2>

```
{ last_name : "Smith", best_score: 3.9 }
```

If you shorten the field named `last_name` to `lname` and the field name `best_score` to `score`, as follows, you could save 9 bytes per document.

```
{ lname : "Smith", score : 3.9 }
```

Shortening field names reduces expressiveness and does not provide considerable benefit on for larger documents and where document overhead is not significant concern. Shorter field names do not reduce the size of indexes, because indexes have a predefined structure.

In general it is not necessary to use short field names.

- Embed documents.

In some cases you may want to embed documents in other documents and save on the per-document overhead.

10.2.11 When should I use GridFS?

For documents in a MongoDB collection, you should always use *GridFS* for storing files larger than 16 MB.

In some situations, storing large files may be more efficient in a MongoDB database than on a system-level filesystem.

- If your filesystem limits the number of files in a directory, you can use GridFS to store as many files as needed.
- When you want to keep your files and metadata automatically synced and deployed across a number of systems and facilities. When using *geographically distributed replica sets* (page 392) MongoDB can distribute files and their metadata automatically to a number of `mongod` instances and facilities.
- When you want to access information from portions of large files without having to load whole files into memory, you can use GridFS to recall sections of files without reading the entire file into memory.

Do not use GridFS if you need to update the content of the entire file atomically. As an alternative you can store multiple versions of each file and specify the current version of the file in the metadata. You can update the metadata field that indicates “latest” status in an atomic update after uploading the new version of the file, and later remove previous versions if needed.

Furthermore, if your files are all smaller the 16 MB BSON Document Size limit, consider storing the file manually within a single document. You may use the `BinData` data type to store the binary data. See your *drivers* (page 92) documentation for details on using `BinData`.

For more information on GridFS, see [GridFS](#) (page 102).

10.2.12 How does MongoDB address SQL or Query injection?

BSON

As a client program assembles a query in MongoDB, it builds a BSON object, not a string. Thus traditional SQL injection attacks are not a problem. More details and some nuances are covered below.

MongoDB represents queries as *BSON* objects. Typically *client libraries* (page 92) provide a convenient, injection free, process to build these objects. Consider the following C++ example:

```
 BSONObj my_query = BSON( "name" << a_name );
auto_ptr<DBClientCursor> cursor = c.query("tutorial.persons", my_query);
```

Here, `my_query` then will have a value such as `{ name : "Joe" }`. If `my_query` contained special characters, for example `,`, `:`, and `{`, the query simply wouldn’t match any documents. For example, users cannot hijack a query and convert it to a delete.

JavaScript

Note: You can disable all server-side execution of JavaScript, by passing the `--noscripting` option on the command line or setting noscripting in a configuration file.

All of the following MongoDB operations permit you to run arbitrary JavaScript expressions directly on the server:

- `$where`
- `db.eval()`
- `mapReduce`
- `group`

You must exercise care in these cases to prevent users from submitting malicious JavaScript.

Fortunately, you can express most queries in MongoDB without JavaScript and for queries that require JavaScript, you can mix JavaScript and non-JavaScript in a single query. Place all the user-supplied fields directly in a *BSON* field and pass JavaScript code to the `$where` field.

- If you need to pass user-supplied values in a `$where` clause, you may escape these values with the `CodeWScope` mechanism. When you set user-submitted values as variables in the scope document, you can avoid evaluating them on the database server.
- If you need to use `db.eval()` with user supplied values, you can either use a `CodeWScope` or you can supply extra arguments to your function. For instance:

```
db.eval(function(userVal){...},
        user_value);
```

This will ensure that your application sends `user_value` to the database server as data rather than code.

Dollar Sign Operator Escaping

Field names in MongoDB's query language have semantic meaning. The dollar sign (i.e `$`) is a reserved character used to represent operators (i.e. `$inc`). Thus, you should ensure that your application's users cannot inject operators into their inputs.

In some cases, you may wish to build a BSON object with a user-provided key. In these situations, keys will need to substitute the reserved `$` and `.` characters. Any character is sufficient, but consider using the Unicode full width equivalents: U+FF04 (i.e. “`”$`”) and U+FF0E (i.e. “`.”`”).

Consider the following example:

```
 BSONObj my_object = BSON( a_key << a_name );
```

The user may have supplied a `$` value in the `a_key` value. At the same time, `my_object` might be `{ $where : "things" }`. Consider the following cases:

- **Insert.** Inserting this into the database does no harm. The insert process does not evaluate the object as a query.

Note: MongoDB client drivers, if properly implemented, check for reserved characters in keys on inserts.

- **Update.** The `update()` operation permits `$` operators in the update argument but does not support the `$where` operator. Still, some users may be able to inject operators that can manipulate a single document only. Therefore your application should escape keys, as mentioned above, if reserved characters are possible.

- **Query** Generally this is not a problem for queries that resemble `{ x : user_obj }`: dollar signs are not top level and have no effect. Theoretically it may be possible for the user to build a query themselves. But checking the user-submitted content for `$` characters in key names may help protect against this kind of injection.

Driver-Specific Issues

See the “[PHP MongoDB Driver Security Notes](#)⁸” page in the PHP driver documentation for more information

10.2.13 How does MongoDB provide concurrency?

MongoDB implements a readers-writer lock. This means that at any one time, only one client may be writing or any number of clients may be reading, but that reading and writing cannot occur simultaneously.

In standalone and *replica sets* the lock’s scope applies to a single `mongod` instance or *primary* instance. In a sharded cluster, locks apply to each individual shard, not to the whole cluster.

For more information, see [FAQ: Concurrency](#) (page 586).

10.2.14 What is the compare order for BSON types?

MongoDB permits documents within a single collection to have fields with different *BSON* types. For instance, the following documents may exist within a single collection.

```
{ x: "string" }  
{ x: 42 }
```

When comparing values of different *BSON* types, MongoDB uses the following comparison order, from lowest to highest:

1. MinKey (internal type)
2. Null
3. Numbers (ints, longs, doubles)
4. Symbol, String
5. Object
6. Array
7. BinData
8. ObjectId
9. Boolean
10. Date, Timestamp
11. Regular Expression
12. MaxKey (internal type)

Note: MongoDB treats some types as equivalent for comparison purposes. For instance, numeric types undergo conversion before comparison.

Consider the following `mongo` example:

⁸<http://us.php.net/manual/en/mongo.security.php>

```
db.test.insert( {x : 3} );
db.test.insert( {x : 2.9} );
db.test.insert( {x : new Date()} );
db.test.insert( {x : true} );

db.test.find().sort({x:1});
{ "_id" : ObjectId("4b0315dce8de6586fb002c7"), "x" : 2.9 }
{ "_id" : ObjectId("4b03154cce8de6586fb002c6"), "x" : 3 }
{ "_id" : ObjectId("4b031566ce8de6586fb002c9"), "x" : true }
{ "_id" : ObjectId("4b031563ce8de6586fb002c8"), "x" : "Tue Nov 17 2009 16:28:03 GMT-0500 (EST)" }
```

The `$type` operator provides access to *BSON type* comparison in the MongoDB query syntax. See the documentation on *BSON types* and the `$type` operator for additional information.

Warning: Storing values of the different types in the same field in a collection is *strongly discouraged*.

See also:

- The [Tailable Cursors](#) (page 71) page for an example of a C++ use of `MinKey`.

10.2.15 How do I query for fields that have null values?

Fields in a document may store `null` values, as in a notional collection, `test`, with the following documents:

```
{ _id: 1, cancelDate: null }
{ _id: 2 }
```

Different query operators treat `null` values differently:

- The `{ cancelDate : null }` query matches documents that either contains the `cancelDate` field whose value is `null` *or* that do not contain the `cancelDate` field:

```
db.test.find( { cancelDate: null } )
```

The query returns both documents:

```
{ "_id" : 1, "cancelDate" : null }
{ "_id" : 2 }
```

- The `{ cancelDate : { $type: 10 } }` query matches documents that contains the `cancelDate` field whose value is `null` *only*; i.e. the value of the `cancelDate` field is of BSON Type Null (i.e. 10):

```
db.test.find( { cancelDate : { $type: 10 } } )
```

The query returns only the document that contains the `null` value:

```
{ "_id" : 1, "cancelDate" : null }
```

- The `{ cancelDate : { $exists: false } }` query matches documents that do not contain the `cancelDate` field:

```
db.test.find( { cancelDate : { $exists: false } } )
```

The query returns only the document that does *not* contain the `cancelDate` field:

```
{ "_id" : 2 }
```

See also:

The reference documentation for the `$type` and `$exists` operators.

10.2.16 Are there any restrictions on the names of Collections?

Collection names can be any UTF-8 string with the following exceptions:

- A collection name should begin with a letter or an underscore.
- The empty string (" ") is not a valid collection name.
- Collection names cannot contain the \$ character. (version 2.2 only)
- Collection names cannot contain the null character: \0
- Do not name a collection using the system. prefix. MongoDB reserves system. for system collections, such as the system.indexes collection.
- The maximum size of a collection name is 128 characters, including the name of the database. However, for maximum flexibility, collections should have names less than 80 characters.

If your collection name includes special characters, such as the underscore character, then to access the collection use the db.getCollection() method or a similar method for your driver⁹.

Example

To create a collection _foo and insert the { a : 1 } document, use the following operation:

```
db.getCollection("foo").insert( { a : 1 } )
```

To perform a query, use the find() method, in as the following:

```
db.getCollection("foo").find()
```

10.2.17 How do I isolate cursors from intervening write operations?

MongoDB cursors can return the same document more than once in some situations.¹⁰ You can use the snapshot() method on a cursor to isolate the operation for a very specific case.

snapshot() traverses the index on the _id field and guarantees that the query will return each document (with respect to the value of the _id field) no more than once.¹¹

The snapshot() does not guarantee that the data returned by the query will reflect a single moment in time *nor* does it provide isolation from insert or delete operations.

Warning:

- You **cannot** use snapshot() with *sharded collections*.
- You **cannot** use snapshot() with sort() or hint() cursor methods.

As an alternative, if your collection has a field or fields that are never modified, you can use a *unique* index on this field or these fields to achieve a similar result as the snapshot(). Query with hint() to explicitly force the query to use that index.

⁹<http://api.mongodb.org/>

¹⁰ As a cursor returns documents other operations may interleave with the query: if some of these operations are *updates* (page 40) that cause the document to move (in the case of a table scan, caused by document growth,) or that change the indexed field on the index used by the query; then the cursor will return the same document more than once.

¹¹ MongoDB does not permit changes to the value of the _id field; it is not possible for a cursor that transverses this index to pass the same document more than once.

10.2.18 When should I embed documents within other documents?

When *modeling data in MongoDB* (page 97), embedding is frequently the choice for:

- “contains” relationships between entities.
- one-to-many relationships when the “many” objects *always* appear with or are viewed in the context of their parents.

You should also consider embedding for performance reasons if you have a collection with a large number of small documents. Nevertheless, if small, separate documents represent the natural model for the data, then you should maintain that model.

If, however, you can group these small documents by some logical relationship *and* you frequently retrieve the documents by this grouping, you might consider “rolling-up” the small documents into larger documents that contain an array of subdocuments. Keep in mind that if you often only need to retrieve a subset of the documents within the group, then “rolling-up” the documents may not provide better performance.

“Rolling up” these small documents into logical groupings means that queries to retrieve a group of documents involve sequential reads and fewer random disk accesses.

Additionally, “rolling up” documents and moving common fields to the larger document benefit the index on these fields. There would be fewer copies of the common fields *and* there would be fewer associated key entries in the corresponding index. See *Index Concepts* (page 314) for more information on indexes.

10.2.19 Where can I learn more about data modeling in MongoDB?

Begin by reading the documents in the *Data Models* (page 95) section. These documents contain a high level introduction to data modeling considerations in addition to practical examples of data models targeted at particular issues.

Additionally, consider the following external resources that provide additional examples:

- Schema Design by Example¹²
- Dynamic Schema Blog Post¹³
- MongoDB Data Modeling and Rails¹⁴
- Ruby Example of Materialized Paths¹⁵
- Sean Cribs Blog Post¹⁶ which was the source for much of the *data-modeling-trees* content.

10.2.20 Can I manually pad documents to prevent moves during updates?

An update can cause a document to move on disk if the document grows in size. To *minimize* document movements, MongoDB uses *padding* (page 55).

You should not have to pad manually because MongoDB adds *padding automatically* (page 55) and can adaptively adjust the amount of padding added to documents to prevent document relocations following updates. You can change the default `paddingFactor` calculation by using the `collMod` command with the `usePowerOf2Sizes` flag. The `usePowerOf2Sizes` flag ensures that MongoDB allocates document space in sizes that are powers of 2, which helps ensure that MongoDB can efficiently reuse space created by document deletion or relocation.

¹²<http://www.10gen.com/presentations/mongodb-melbourne-2012/schema-design-example>

¹³<http://dmerr.tumblr.com/post/6633338010/schemaless>

¹⁴<http://docs.mongodb.org/ecosystem/tutorial/model-data-for-ruby-on-rails/>

¹⁵<http://github.com/banker/newsmonger/blob/master/app/models/comment.rb>

¹⁶<http://seancribbs.com/tech/2009/09/28/modeling-a-tree-in-a-document-database>

However, if you must pad a document manually, you can add a temporary field to the document and then \$unset the field, as in the following example.

Warning: Do not manually pad documents in a capped collection. Applying manual padding to a document in a capped collection can break replication. Also, the padding is not preserved if you re-sync the MongoDB instance.

```
var myTempPadding = [ "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa",
                      "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa",
                      "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa",
                      "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"];

db.myCollection.insert( { _id: 5, paddingField: myTempPadding } );

db.myCollection.update( { _id: 5 },
                       { $unset: { paddingField: "" } }
                     )

db.myCollection.update( { _id: 5 },
                       { $set: { realField: "Some text that I might have needed padding for" } }
                     )
```

See also:

[Padding Factor](#) (page 55)

10.3 FAQ: The mongo Shell

Frequently Asked Questions:

- How can I enter multi-line operations in the mongo shell? (page 584)
- How can I access different databases temporarily? (page 585)
- Does the mongo shell support tab completion and other keyboard shortcuts? (page 585)
- How can I customize the mongo shell prompt? (page 585)
- Can I edit long shell operations with an external text editor? (page 586)

10.3.1 How can I enter multi-line operations in the mongo shell?

If you end a line with an open parenthesis ('('), an open brace ('{'), or an open bracket ('['), then the subsequent lines start with ellipsis ("...") until you enter the corresponding closing parenthesis (')'), the closing brace ('}') or the closing bracket (']'). The mongo shell waits for the closing parenthesis, closing brace, or the closing bracket before evaluating the code, as in the following example:

```
> if ( x > 0 ) {
... count++;
... print (x);
... }
```

You can exit the line continuation mode if you enter two blank lines, as in the following example:

```
> if (x > 0
...
...
>
```

10.3.2 How can I access different databases temporarily?

You can use `db.getSiblingDB()` method to access another database without switching databases, as in the following example which first switches to the `test` database and then accesses the `sampleDB` database from the `test` database:

```
use test

db.getSiblingDB('sampleDB').getCollectionNames();
```

10.3.3 Does the mongo shell support tab completion and other keyboard shortcuts?

The mongo shell supports keyboard shortcuts. For example,

- Use the up/down arrow keys to scroll through command history. See `.dbshell` documentation for more information on the `.dbshell` file.
- Use `<Tab>` to autocomplete or to list the completion possibilities, as in the following example which uses `<Tab>` to complete the method name starting with the letter '`c`' :

```
db.myCollection.c<Tab>
```

Because there are many collection methods starting with the letter '`c`', the `<Tab>` will list the various methods that start with '`c`'.

For a full list of the shortcuts, see *Shell Keyboard Shortcuts*

10.3.4 How can I customize the mongo shell prompt?

New in version 1.9.

You can change the mongo shell prompt by setting the `prompt` variable. This makes it possible to display additional information in the prompt.

Set `prompt` to any string or arbitrary JavaScript code that returns a string, consider the following examples:

- Set the shell prompt to display the hostname and the database issued:

```
var host = db.serverStatus().host;
var prompt = function() { return db+"@"+host+">> ";
```

The mongo shell prompt should now reflect the new prompt:

```
test@my-machine.local>
```

- Set the shell prompt to display the database statistics:

```
var prompt = function() {
    return "Uptime:"+db.serverStatus().uptime+" Documents:"+db.stats().objects+">> "
}
```

The mongo shell prompt should now reflect the new prompt:

```
Uptime:1052 Documents:25024787 >
```

You can add the logic for the prompt in the `.mongorc.js` file to set the prompt each time you start up the mongo shell.

10.3.5 Can I edit long shell operations with an external text editor?

You can use your own editor in the mongo shell by setting the EDITOR environment variable before starting the mongo shell. Once in the mongo shell, you can edit with the specified editor by typing `edit <variable>` or `edit <function>`, as in the following example:

1. Set the EDITOR variable from the command line prompt:

```
EDITOR=vim
```

2. Start the mongo shell:

```
mongo
```

3. Define a function myFunction:

```
function myFunction () { }
```

4. Edit the function using your editor:

```
edit myFunction
```

The command should open the vim edit session. Remember to save your changes.

5. Type myFunction to see the function definition:

```
myFunction
```

The result should be the changes from your saved edit:

```
function myFunction() {
    print("This was edited");
}
```

10.4 FAQ: Concurrency

Frequently Asked Questions:

- What type of locking does MongoDB use? (page 587)
- How granular are locks in MongoDB? (page 587)
- How do I see the status of locks on my mongod instances? (page 587)
- Does a read or write operation ever yield the lock? (page 587)
- Which operations lock the database? (page 588)
- Which administrative commands lock the database? (page 588)
- Does a MongoDB operation ever lock more than one database? (page 589)
- How does sharding affect concurrency? (page 589)
- How does concurrency affect a replica set primary? (page 589)
- How does concurrency affect secondaries? (page 589)
- What kind of concurrency does MongoDB provide for JavaScript operations? (page 590)

Changed in version 2.2.

MongoDB allows multiple clients to read and write a single corpus of data using a locking system to ensure that all clients receive a consistent view of the data *and* to prevent multiple applications from modifying the exact same pieces of data at the same time. Locks help guarantee that all writes to a single document occur either in full or not at all.

See also:

Presentation on Concurrency and Internals in 2.2¹⁷

10.4.1 What type of locking does MongoDB use?

MongoDB uses a readers-writer¹⁸ lock that allows concurrent reads access to a database but gives exclusive access to a single write operation.

When a read lock exists, many read operations may use this lock. However, when a write lock exists, a single write operation holds the lock exclusively, and no other read *or* write operations may share the lock.

Locks are “writer greedy,” which means writes have preference over reads. When both a read and write are waiting for a lock, MongoDB grants the lock to the write.

10.4.2 How granular are locks in MongoDB?

Changed in version 2.2.

Beginning with version 2.2, MongoDB implements locks on a per-database basis for most read and write operations. Some global operations, typically short lived operations involving multiple databases, still require a global “instance” wide lock. Before 2.2, there is only one “global” lock per mongod instance.

For example, if you have six databases and one takes a write lock, the other five are still available for read and write.

10.4.3 How do I see the status of locks on my mongod instances?

For reporting on lock utilization information on locks, use any of the following methods:

- `db.serverStatus()`,
- `db.currentOp()`,
- `mongotop`,
- `mongostat`, and/or
- the [MongoDB Management Service \(MMS\)](#)¹⁹

Specifically, the `locks` document in the output of `serverStatus`, or the `locks` field in the current operation reporting provides insight into the type of locks and amount of lock contention in your mongod instance.

To terminate an operation, use `db.killOp()`.

10.4.4 Does a read or write operation ever yield the lock?

In some situations, read and write operations can yield their locks.

Long running read and write operations, such as queries, updates, and deletes, yield under many conditions. In MongoDB 2.0, operations yielded based on time slices and the number of operations waiting for the actively held lock. After 2.2, more adaptive algorithms allow operations to yield based on predicted disk access (i.e. page faults).

¹⁷<http://www.mongodb.com/presentations/concurrency-internals-mongodb-2-2>

¹⁸ You may be familiar with a “readers-writer” lock as “multi-reader” or “shared exclusive” lock. See the Wikipedia page on [Readers-Writer Locks](http://en.wikipedia.org/wiki/Readers%20writer_lock) (http://en.wikipedia.org/wiki/Readers%20writer_lock) for more information.

¹⁹<http://mms.mongodb.com/>

New in version 2.0: Read and write operations will yield their locks if the mongod receives a *page fault* or fetches data that is unlikely to be in memory. Yielding allows other operations that only need to access documents that are already in memory to complete while mongod loads documents into memory.

Additionally, write operations that affect multiple documents (i.e. `update()` with the `multi` parameter,) will yield periodically to allow read operations during these long write operations. Similarly, long running read locks will yield periodically to ensure that write operations have the opportunity to complete.

Changed in version 2.2: The use of yielding expanded greatly in MongoDB 2.2. Including the “yield for page fault.” MongoDB tracks the contents of memory and predicts whether data is available before performing a read. If MongoDB predicts that the data is not in memory a read operation yields its lock while MongoDB loads the data to memory. Once data is available in memory, the read will reacquire the lock to complete the operation.

10.4.5 Which operations lock the database?

Changed in version 2.2.

The following table lists common database operations and the types of locks they use.

Operation	Lock Type
Issue a query	Read lock
Get more data from a <i>cursor</i>	Read lock
Insert data	Write lock
Remove data	Write lock
Update data	Write lock
<i>Map-reduce</i>	Read lock and write lock, unless operations are specified as non-atomic. Portions of map-reduce jobs can run concurrently.
Create an index	Building an index in the foreground, which is the default, locks the database for extended periods of time.
<code>db.eval()</code>	Write lock. <code>db.eval()</code> blocks all other JavaScript processes.
<code>eval</code>	Write lock. If used with the <code>nolock</code> lock option, the <code>eval</code> option does not take a write lock and cannot write data to the database.
<code>aggregate()</code>	Read lock

10.4.6 Which administrative commands lock the database?

Certain administrative commands can exclusively lock the database for extended periods of time. In some deployments, for large databases, you may consider taking the mongod instance offline so that clients are not affected. For example, if a mongod is part of a *replica set*, take the mongod offline and let other members of the set service load while maintenance is in progress.

The following administrative operations require an exclusive (i.e. write) lock on the database for extended periods:

- `db.collection.ensureIndex()`, when issued *without* setting `background` to `true`,
- `reIndex`,
- `compact`,
- `db.repairDatabase()`,
- `db.createCollection()`, when creating a very large (i.e. many gigabytes) capped collection,
- `db.collection.validate()`, and
- `db.copyDatabase()`. This operation may lock all databases. See [Does a MongoDB operation ever lock more than one database? \(page 589\)](#).

The following administrative commands lock the database but only hold the lock for a very short time:

- `db.collection.dropIndex()`,
- `db.getLastError()`,
- `db.isMaster()`,
- `rs.status()` (i.e. `replSetGetStatus`),
- `db.serverStatus()`,
- `db.auth()`, and
- `db.addUser()`.

10.4.7 Does a MongoDB operation ever lock more than one database?

The following MongoDB operations lock multiple databases:

- `db.copyDatabase()` must lock the entire `mongod` instance at once.
- *Journaling*, which is an internal operation, locks all databases for short intervals. All databases share a single journal.
- *User authentication* (page 235) locks the `admin` database as well as the database the user is accessing.
- All writes to a replica set's *primary* lock both the database receiving the writes and then the `local` database for a short time. The lock for the `local` database allows the `mongod` to write to the primary's *oplog* and accounts for a small portion of the total time of the operation.

10.4.8 How does sharding affect concurrency?

Sharding improves concurrency by distributing collections over multiple `mongod` instances, allowing shard servers (i.e. `mongos` processes) to perform any number of operations concurrently to the various downstream `mongod` instances.

Each `mongod` instance is independent of the others in the shard cluster and uses the MongoDB *readers-writer lock* (page 587). The operations on one `mongod` instance do not block the operations on any others.

10.4.9 How does concurrency affect a replica set primary?

In *replication*, when MongoDB writes to a collection on the *primary*, MongoDB also writes to the primary's *oplog*, which is a special collection in the `local` database. Therefore, MongoDB must lock both the collection's database and the `local` database. The `mongod` must lock both databases at the same time keep both data consistent and ensure that write operations, even with replication, are “all-or-nothing” operations.

10.4.10 How does concurrency affect secondaries?

In *replication*, MongoDB does not apply writes serially to *secondaries*. Secondaries collect oplog entries in batches and then apply those batches in parallel. Secondaries do not allow reads while applying the write operations, and apply write operations in the order that they appear in the oplog.

MongoDB can apply several writes in parallel on replica set secondaries, in two phases:

1. During the first *prefer* phase, under a read lock, the `mongod` ensures that all documents affected by the operations are in memory. During this phase, other clients may execute queries against this member.

2. A thread pool using write locks applies all write operations in the batch as part of a coordinated write phase.

10.4.11 What kind of concurrency does MongoDB provide for JavaScript operations?

Changed in version 2.4: The V8 JavaScript engine added in 2.4 allows multiple JavaScript operations to run at the same time. Prior to 2.4, a single `mongod` could only run a *single* JavaScript operation at once.

10.5 FAQ: Sharding with MongoDB

Frequently Asked Questions:

- Is sharding appropriate for a new deployment? (page 590)
- How does sharding work with replication? (page 591)
- Can I change the shard key after sharding a collection? (page 591)
- What happens to unsharded collections in sharded databases? (page 591)
- How does MongoDB distribute data across shards? (page 591)
- What happens if a client updates a document in a chunk during a migration? (page 591)
- What happens to queries if a shard is inaccessible or slow? (page 592)
- How does MongoDB distribute queries among shards? (page 592)
- How does MongoDB sort queries in sharded environments? (page 592)
- How does MongoDB ensure unique `_id` field values when using a shard key *other* than `_id`? (page 592)
- I've enabled sharding and added a second shard, but all the data is still on one server. Why? (page 592)
- Is it safe to remove old files in the `moveChunk` directory? (page 593)
- How does `mongos` use connections? (page 593)
- Why does `mongos` hold connections open? (page 593)
- Where does MongoDB report on connections used by `mongos`? (page 593)
- What does `writebacklisten` in the log mean? (page 593)
- How should administrators deal with failed migrations? (page 593)
- What is the process for moving, renaming, or changing the number of config servers? (page 594)
- When do the `mongos` servers detect config server changes? (page 594)
- Is it possible to quickly update `mongos` servers after updating a replica set configuration? (page 594)
- What does the `maxConns` setting on `mongos` do? (page 594)
- How do indexes impact queries in sharded systems? (page 594)
- Can shard keys be randomly generated? (page 594)
- Can shard keys have a non-uniform distribution of values? (page 595)
- Can you shard on the `_id` field? (page 595)
- What do `moveChunk` commit failed errors mean? (page 595)
- How does draining a shard affect the balancing of uneven chunk distribution? (page 595)

This document answers common questions about horizontal scaling using MongoDB's *sharding*.

If you don't find the answer you're looking for, check the *complete list of FAQs* (page 571) or post your question to the MongoDB User Mailing List²⁰.

10.5.1 Is sharding appropriate for a new deployment?

Sometimes.

²⁰<https://groups.google.com/forum/?fromgroups#!forum/mongodb-user>

If your data set fits on a single server, you should begin with an unsharded deployment.

Converting an unsharded database to a *sharded cluster* is easy and seamless, so there is *little advantage* in configuring sharding while your data set is small.

Still, all production deployments should use *replica sets* to provide high availability and disaster recovery.

10.5.2 How does sharding work with replication?

To use replication with sharding, deploy each *shard* as a *replica set*.

10.5.3 Can I change the shard key after sharding a collection?

No.

There is no automatic support in MongoDB for changing a shard key after sharding a collection. This reality underscores the importance of choosing a good *shard key* (page 499). If you *must* change a shard key after sharding a collection, the best option is to:

- dump all data from MongoDB into an external format.
- drop the original sharded collection.
- configure sharding using a more ideal shard key.
- *pre-split* (page 544) the shard key range to ensure initial even distribution.
- restore the dumped data into MongoDB.

See `shardCollection`, `sh.shardCollection()`, the *Shard Key* (page 499), *Deploy a Sharded Cluster* (page 514), and SERVER-4000²¹ for more information.

10.5.4 What happens to unsharded collections in sharded databases?

In the current implementation, all databases in a *sharded cluster* have a “primary shard.” All unsharded collection within that database will reside on the same shard.

10.5.5 How does MongoDB distribute data across shards?

Sharding must be specifically enabled on a collection. After enabling sharding on the collection, MongoDB will assign various ranges of collection data to the different shards in the cluster. The cluster automatically corrects imbalances between shards by migrating ranges of data from one shard to another.

10.5.6 What happens if a client updates a document in a chunk during a migration?

The `mongos` routes the operation to the “old” shard, where it will succeed immediately. Then the *shard* `mongod` instances will replicate the modification to the “new” shard before the *sharded cluster* updates that chunk’s “ownership,” which effectively finalizes the migration process.

²¹<https://jira.mongodb.org/browse/SERVER-4000>

10.5.7 What happens to queries if a shard is inaccessible or slow?

If a *shard* is inaccessible or unavailable, queries will return with an error.

However, a client may set the `partial` query bit, which will then return results from all available shards, regardless of whether a given shard is unavailable.

If a shard is responding slowly, `mongos` will merely wait for the shard to return results.

10.5.8 How does MongoDB distribute queries among shards?

Changed in version 2.0.

The exact method for distributing queries to *shards* in a *cluster* depends on the nature of the query and the configuration of the sharded cluster. Consider a sharded collection, using the *shard key* `user_id`, that has `last_login` and `email` attributes:

- For a query that selects one or more values for the `user_id` key:

`mongos` determines which shard or shards contains the relevant data, based on the cluster metadata, and directs a query to the required shard or shards, and returns those results to the client.

- For a query that selects `user_id` and also performs a sort:

`mongos` can make a straightforward translation of this operation into a number of queries against the relevant shards, ordered by `user_id`. When the sorted queries return from all shards, the `mongos` merges the sorted results and returns the complete result to the client.

- For queries that select on `last_login`:

These queries must run on all shards: `mongos` must parallelize the query over the shards and perform a merge-sort on the `email` of the documents found.

10.5.9 How does MongoDB sort queries in sharded environments?

If you call the `cursor.sort()` method on a query in a sharded environment, the `mongod` for each shard will sort its results, and the `mongos` merges each shard's results before returning them to the client.

10.5.10 How does MongoDB ensure unique `_id` field values when using a shard key other than `_id`?

If you do not use `_id` as the shard key, then your application/client layer must be responsible for keeping the `_id` field unique. It is problematic for collections to have duplicate `_id` values.

If you're not sharding your collection by the `_id` field, then you should be sure to store a globally unique identifier in that field. The default [BSON ObjectID](#) (page 127) works well in this case.

10.5.11 I've enabled sharding and added a second shard, but all the data is still on one server. Why?

First, ensure that you've declared a *shard key* for your collection. Until you have configured the shard key, MongoDB will not create *chunks*, and *sharding* will not occur.

Next, keep in mind that the default chunk size is 64 MB. As a result, in most situations, the collection needs to have at least 64 MB of data before a migration will occur.

Additionally, the system which balances chunks among the servers attempts to avoid superfluous migrations. Depending on the number of shards, your shard key, and the amount of data, systems often require at least 10 chunks of data to trigger migrations.

You can run `db.printShardingStatus()` to see all the chunks present in your cluster.

10.5.12 Is it safe to remove old files in the `moveChunk` directory?

Yes. `mongod` creates these files as backups during normal *shard* balancing operations.

Once these migrations are complete, you may delete these files.

10.5.13 How does `mongos` use connections?

Each client maintains a connection to a `mongos` instance. Each `mongos` instance maintains a pool of connections to the members of a replica set supporting the sharded cluster. Clients use connections between `mongos` and `mongod` instances one at a time. Requests are not multiplexed or pipelined. When client requests complete, the `mongos` returns the connection to the pool.

See the [System Resource Utilization](#) (page 220) section of the [UNIX ulimit Settings](#) (page 219) document.

10.5.14 Why does `mongos` hold connections open?

`mongos` uses a set of connection pools to communicate with each *shard*. These pools do not shrink when the number of clients decreases.

This can lead to an unused `mongos` with a large number of open connections. If the `mongos` is no longer in use, it is safe to restart the process to close existing connections.

10.5.15 Where does MongoDB report on connections used by `mongos`?

Connect to the `mongos` with the `mongo` shell, and run the following command:

```
db._adminCommand("connPoolStats");
```

10.5.16 What does `writebacklisten` in the log mean?

The writeback listener is a process that opens a long poll to relay writes back from a `mongod` or `mongos` after migrations to make sure they have not gone to the wrong server. The writeback listener sends writes back to the correct server if necessary.

These messages are a key part of the sharding infrastructure and should not cause concern.

10.5.17 How should administrators deal with failed migrations?

Failed migrations require no administrative intervention. Chunk moves are consistent and deterministic.

If a migration fails to complete for some reason, the *cluster* will retry the operation. When the migration completes successfully, the data will reside only on the new shard.

10.5.18 What is the process for moving, renaming, or changing the number of config servers?

See *Sharded Cluster Tutorials* (page 513) for information on migrating and replacing config servers.

10.5.19 When do the mongos servers detect config server changes?

mongos instances maintain a cache of the *config database* that holds the metadata for the *sharded cluster*. This metadata includes the mapping of *chunks* to *shards*.

mongos updates its cache lazily by issuing a request to a shard and discovering that its metadata is out of date. There is no way to control this behavior from the client, but you can run the `flushRouterConfig` command against any mongos to force it to refresh its cache.

10.5.20 Is it possible to quickly update mongos servers after updating a replica set configuration?

The mongos instances will detect these changes without intervention over time. However, if you want to force the mongos to reload its configuration, run the `flushRouterConfig` command against each mongos directly.

10.5.21 What does the maxConns setting on mongos do?

The `maxConns` option limits the number of connections accepted by mongos.

If your client driver or application creates a large number of connections but allows them to time out rather than closing them explicitly, then it might make sense to limit the number of connections at the mongos layer.

Set `maxConns` to a value slightly higher than the maximum number of connections that the client creates, or the maximum size of the connection pool. This setting prevents the mongos from causing connection spikes on the individual *shards*. Spikes like these may disrupt the operation and memory allocation of the *sharded cluster*.

10.5.22 How do indexes impact queries in sharded systems?

If the query does not include the *shard key*, the mongos must send the query to all shards as a “scatter/gather” operation. Each shard will, in turn, use *either* the shard key index or another more efficient index to fulfill the query.

If the query includes multiple sub-expressions that reference the fields indexed by the shard key *and* the secondary index, the mongos can route the queries to a specific shard and the shard will use the index that will allow it to fulfill most efficiently. See [this presentation](#)²² for more information.

10.5.23 Can shard keys be randomly generated?

Shard keys can be random. Random keys ensure optimal distribution of data across the cluster.

Sharded clusters, attempt to route queries to *specific shards* when queries include the shard key as a parameter, because these directed queries are more efficient. In many cases, random keys can make it difficult to direct queries to specific shards.

²²<http://www.slideshare.net/mongodb/how-queries-work-with-sharding>

10.5.24 Can shard keys have a non-uniform distribution of values?

Yes. There is no requirement that documents be evenly distributed by the shard key.

However, documents that have the shard key *must* reside in the same *chunk* and therefore on the same server. If your sharded data set has too many documents with the exact same shard key you will not be able to distribute *those* documents across your sharded cluster.

10.5.25 Can you shard on the `_id` field?

You can use any field for the shard key. The `_id` field is a common shard key.

Be aware that `ObjectID()` values, which are the default value of the `_id` field, increment as a timestamp. As a result, when used as a shard key, all new documents inserted into the collection will initially belong to the same chunk on a single shard. Although the system will eventually divide this chunk and migrate its contents to distribute data more evenly, at any moment the cluster can only direct insert operations at a single shard. This can limit the throughput of inserts. If most of your write operations are updates, this limitation should not impact your performance. However, if you have a high insert volume, this may be a limitation.

To address this issue, MongoDB 2.4 provides *hashed shard keys* (page 500).

10.5.26 What do `moveChunk` commit failed errors mean?

Consider the following error message:

```
ERROR: moveChunk commit failed: version is at <n>|<nn> instead of <N>|<NN>" and "ERROR: TERMINATING"
```

mongod issues this message if, during a *chunk migration* (page 510), the *shard* could not connect to the *config database* to update chunk information at the end of the migration process. If the shard cannot update the config database after `moveChunk`, the cluster will have an inconsistent view of all chunks. In these situations, the *primary* member of the shard will terminate itself to prevent data inconsistency. If the *secondary* member can access the config database, the shard's data will be accessible after an election. Administrators will need to resolve the chunk migration failure independently.

If you encounter this issue, contact the [MongoDB User Group²³](#) or MongoDB support to address this issue.

10.5.27 How does draining a shard affect the balancing of uneven chunk distribution?

The sharded cluster balancing process controls both migrating chunks from decommissioned shards (i.e. draining,) and normal cluster balancing activities. Consider the following behaviors for different versions of MongoDB in situations where you remove a shard in a cluster with an uneven chunk distribution:

- After MongoDB 2.2, the balancer first removes the chunks from the draining shard and then balances the remaining uneven chunk distribution.
- Before MongoDB 2.2, the balancer handles the uneven chunk distribution and *then* removes the chunks from the draining shard.

²³<http://groups.google.com/group/mongodb-user>

10.6 FAQ: Replica Sets and Replication in MongoDB

Frequently Asked Questions:

- What kinds of replication does MongoDB support? (page 596)
- What do the terms “primary” and “master” mean? (page 596)
- What do the terms “secondary” and “slave” mean? (page 596)
- How long does replica set failover take? (page 597)
- Does replication work over the Internet and WAN connections? (page 597)
- Can MongoDB replicate over a “noisy” connection? (page 597)
- What is the preferred replication method: master/slave or replica sets? (page 597)
- What is the preferred replication method: replica sets or replica pairs? (page 597)
- Why use journaling if replication already provides data redundancy? (page 597)
- Are write operations durable if write concern does not acknowledge writes? (page 598)
- How many arbiters do replica sets need? (page 598)
- What information do arbiters exchange with the rest of the replica set? (page 598)
- Which members of a replica set vote in elections? (page 599)
- Do hidden members vote in replica set elections? (page 599)
- Is it normal for replica set members to use different amounts of disk space? (page 599)

This document answers common questions about database replication in MongoDB.

If you don’t find the answer you’re looking for, check the [complete list of FAQs](#) (page 571) or post your question to the MongoDB User Mailing List²⁴.

10.6.1 What kinds of replication does MongoDB support?

MongoDB supports master-slave replication and a variation on master-slave replication known as replica sets. Replica sets are the recommended replication topology.

10.6.2 What do the terms “primary” and “master” mean?

Primary and *master* nodes are the nodes that can accept writes. MongoDB’s replication is “single-master:” only one node can accept write operations at a time.

In a replica set, if the current “primary” node fails or becomes inaccessible, the other members can autonomously *elect* one of the other members of the set to be the new “primary”.

By default, clients send all reads to the primary; however, *read preference* is configurable at the client level on a per-connection basis, which makes it possible to send reads to secondary nodes instead.

10.6.3 What do the terms “secondary” and “slave” mean?

Secondary and *slave* nodes are read-only nodes that replicate from the *primary*.

Replication operates by way of an *oplog*, from which secondary/slave members apply new operations to themselves. This replication process is asynchronous, so secondary/slave nodes may not always reflect the latest writes to the primary. But usually, the gap between the primary and secondary nodes is just few milliseconds on a local network connection.

²⁴<https://groups.google.com/forum/?fromgroups#!forum/mongodb-user>

10.6.4 How long does replica set failover take?

It varies, but a replica set will select a new primary within a minute.

It may take 10-30 seconds for the members of a *replica set* to declare a *primary* inaccessible. This triggers an *election*. During the election, the cluster is unavailable for writes.

The election itself may take another 10-30 seconds.

Note: *Eventually consistent* reads, like the ones that will return from a replica set are only possible with a *write concern* that permits reads from *secondary* members.

10.6.5 Does replication work over the Internet and WAN connections?

Yes.

For example, a deployment may maintain a *primary* and *secondary* in an East-coast data center along with a *secondary* member for disaster recovery in a West-coast data center.

See also:

[Deploy a Geographically Redundant Replica Set](#) (page 421)

10.6.6 Can MongoDB replicate over a “noisy” connection?

Yes, but not without connection failures and the obvious latency.

Members of the set will attempt to reconnect to the other members of the set in response to networking flaps. This does not require administrator intervention. However, if the network connections among the nodes in the replica set are very slow, it might not be possible for the members of the node to keep up with the replication.

If the TCP connection between the secondaries and the *primary* instance breaks, a *replica set* will automatically elect one of the *secondary* members of the set as primary.

10.6.7 What is the preferred replication method: master/slave or replica sets?

New in version 1.8.

Replica sets are the preferred *replication* mechanism in MongoDB. However, if your deployment requires more than 12 nodes, you must use master/slave replication.

10.6.8 What is the preferred replication method: replica sets or replica pairs?

Deprecated since version 1.6.

Replica sets replaced *replica pairs* in version 1.6. *Replica sets* are the preferred *replication* mechanism in MongoDB.

10.6.9 Why use journaling if replication already provides data redundancy?

Journaling facilitates faster crash recovery. Prior to journaling, crashes often required database repairs or full data resync. Both were slow, and the first was unreliable.

Journaling is particularly useful for protection against power failures, especially if your replica set resides in a single data center or power circuit.

When a *replica set* runs with journaling, `mongod` instances can safely restart without any administrator intervention.

Note: Journaling requires some resource overhead for write operations. Journaling has no effect on read performance, however.

Journaling is enabled by default on all 64-bit builds of MongoDB v2.0 and greater.

10.6.10 Are write operations durable if write concern does not acknowledge writes?

Yes.

However, if you want confirmation that a given write has arrived at the server, use [write concern](#) (page 44). The `getLastError` command provides the facility for write concern. However, after the [default write concern change](#) (page 654), the default write concern acknowledges all write operations, and unacknowledged writes must be explicitly configured. See the [MongoDB Drivers and Client Libraries](#) (page 92) documentation for your driver for more information.

10.6.11 How many arbiters do replica sets need?

Some configurations do not require any *arbiter* instances. Arbiters vote in *elections* for *primary* but do not replicate the data like *secondary* members.

Replica sets require a majority of the remaining nodes present to elect a primary. Arbiters allow you to construct this majority without the overhead of adding replicating nodes to the system.

There are many possible replica set [architectures](#) (page 386).

If you have a three node replica set, you don't need an arbiter.

But a common configuration consists of two replicating nodes, one of which is *primary* and the other is *secondary*, as well as an arbiter for the third node. This configuration makes it possible for the set to elect a primary in the event of a failure without requiring three replicating nodes.

You may also consider adding an arbiter to a set if it has an equal number of nodes in two facilities and network partitions between the facilities are possible. In these cases, the arbiter will break the tie between the two facilities and allow the set to elect a new primary.

See also:

[Replica Set Deployment Architectures](#) (page 386)

10.6.12 What information do arbiters exchange with the rest of the replica set?

Arbiters never receive the contents of a collection but do exchange the following data with the rest of the replica set:

- Credentials used to authenticate the arbiter with the replica set. All MongoDB processes within a replica set use keyfiles. These exchanges are encrypted.
- Replica set configuration data and voting data. This information is not encrypted. Only credential exchanges are encrypted.

If your MongoDB deployment uses SSL, then all communications between arbiters and the other members of the replica set are secure. See the documentation for [Connect to MongoDB with SSL](#) (page 248) for more information. Run all arbiters on secure networks, as with all MongoDB components.

See

The overview of *Arbiter Members of Replica Sets* (page ??).

10.6.13 Which members of a replica set vote in elections?

All members of a replica set, unless the value of `votes` (page 476) is equal to 0, vote in elections. This includes all *delayed* (page 383), *hidden* (page 383) and *secondary-only* (page 382) members, as well as the *arbiters* (page ??).

Additionally, the state of the voting members also determine whether the member can vote. Only voting members in the following states are eligible to vote:

- PRIMARY
- SECONDARY
- RECOVERING
- ARBITER
- ROLLBACK

See also:

Replica Set Elections (page 393)

10.6.14 Do hidden members vote in replica set elections?

Hidden members (page 383) of *replica sets* do vote in elections. To exclude a member from voting in an *election*, change the value of the member's `votes` (page 476) configuration to 0.

See also:

Replica Set Elections (page 393)

10.6.15 Is it normal for replica set members to use different amounts of disk space?

Yes.

Factors including: different oplog sizes, different levels of storage fragmentation, and MongoDB's data file pre-allocation can lead to some variation in storage utilization between nodes. Storage use disparities will be most pronounced when you add members at different times.

10.7 FAQ: MongoDB Storage

Frequently Asked Questions:

- What are memory mapped files? (page 600)
- How do memory mapped files work? (page 600)
- How does MongoDB work with memory mapped files? (page 600)
- What are page faults? (page 600)
- What is the difference between soft and hard page faults? (page 601)
- What tools can I use to investigate storage use in MongoDB? (page 601)
- What is the working set? (page 601)
- Why are the files in my data directory larger than the data in my database? (page 601)
- How can I check the size of a collection? (page 602)
- How can I check the size of indexes? (page 602)
- How do I know when the server runs out of disk space? (page 603)

This document addresses common questions regarding MongoDB's storage system.

If you don't find the answer you're looking for, check the *complete list of FAQs* (page 571) or post your question to the MongoDB User Mailing List²⁵.

10.7.1 What are memory mapped files?

A memory-mapped file is a file with data that the operating system places in memory by way of the `mmap()` system call. `mmap()` thus *maps* the file to a region of virtual memory. Memory-mapped files are the critical piece of the storage engine in MongoDB. By using memory mapped files MongoDB can treat the contents of its data files as if they were in memory. This provides MongoDB with an extremely fast and simple method for accessing and manipulating data.

10.7.2 How do memory mapped files work?

Memory mapping assigns files to a block of virtual memory with a direct byte-for-byte correlation. Once mapped, the relationship between file and memory allows MongoDB to interact with the data in the file as if it were memory.

10.7.3 How does MongoDB work with memory mapped files?

MongoDB uses memory mapped files for managing and interacting with all data. MongoDB memory maps data files to memory as it accesses documents. Data that isn't accessed is *not* mapped to memory.

10.7.4 What are page faults?

Page faults will occur if you're attempting to access part of a memory-mapped file that *isn't* in memory.

If there is free memory, then the operating system can find the page on disk and load it to memory directly. However, if there is no free memory, the operating system must:

- find a page in memory that is stale or no longer needed, and write the page to disk.
- read the requested page from disk and load it into memory.

This process, particularly on an active system can take a long time, particularly in comparison to reading a page that is already in memory.

²⁵<https://groups.google.com/forum/?fromgroups#!forum/mongodb-user>

10.7.5 What is the difference between soft and hard page faults?

Page faults occur when MongoDB needs access to data that isn't currently in active memory. A “hard” page fault refers to situations when MongoDB must access a disk to access the data. A “soft” page fault, by contrast, merely moves memory pages from one list to another, such as from an operating system file cache. In production, MongoDB will rarely encounter soft page faults.

10.7.6 What tools can I use to investigate storage use in MongoDB?

The `db.stats()` method in the mongo shell, returns the current state of the “active” database. The `dbStats` command document describes the fields in the `db.stats()` output.

10.7.7 What is the working set?

Working set represents the total body of data that the application uses in the course of normal operation. Often this is a subset of the total data size, but the specific size of the working set depends on actual moment-to-moment use of the database.

If you run a query that requires MongoDB to scan every document in a collection, the working set will expand to include every document. Depending on physical memory size, this may cause documents in the working set to “page out,” or to be removed from physical memory by the operating system. The next time MongoDB needs to access these documents, MongoDB may incur a hard page fault.

If you run a query that requires MongoDB to scan every *document* in a collection, the working set includes every active document in memory.

For best performance, the majority of your *active* set should fit in RAM.

10.7.8 Why are the files in my data directory larger than the data in my database?

The data files in your data directory, which is the `/data/db` directory in default configurations, might be larger than the data set inserted into the database. Consider the following possible causes:

- Preallocated data files.

In the data directory, MongoDB preallocates data files to a particular size, in part to prevent file system fragmentation. MongoDB names the first data file `<datbasename>.0`, the next `<datbasename>.1`, etc. The first file `mongod` allocates is 64 megabytes, the next 128 megabytes, and so on, up to 2 gigabytes, at which point all subsequent files are 2 gigabytes. The data files include files with allocated space but that hold no data. `mongod` may allocate a 1 gigabyte data file that may be 90% empty. For most larger databases, unused allocated space is small compared to the database.

On Unix-like systems, `mongod` preallocates an additional data file and initializes the disk space to 0. Preallocating data files in the background prevents significant delays when a new database file is next allocated.

You can disable preallocation with the `noprealloc` run time option. However `noprealloc` is **not** intended for use in production environments: only use `noprealloc` for testing and with small data sets where you frequently drop databases.

On Linux systems you can use `hdparm` to get an idea of how costly allocation might be:

```
time hdparm --fallocate $((1024*1024)) testfile
```

- The *oplog*.

If this `mongod` is a member of a replica set, the data directory includes the `oplog.rs` file, which is a preallocated *capped collection* in the `local` database. The default allocation is approximately 5% of disk space on 64-bit installations, see [Oplog Sizing](#) (page 406) for more information. In most cases, you should not need to resize the oplog. However, if you do, see [Change the Size of the Oplog](#) (page 441).

- The *journal*.

The data directory contains the journal files, which store write operations on disk prior to MongoDB applying them to databases. See [Journaling Mechanics](#) (page 229).

- Empty records.

MongoDB maintains lists of empty records in data files when deleting documents and collections. MongoDB can reuse this space, but will never return this space to the operating system.

To de-fragment allocated storage, use `compact`, which de-fragments allocated space which allows. By defragmenting storage, MongoDB can effectively use the allocated space. `compact` requires up to 2 gigabytes of extra disk space to run. Do not use `compact` if you are critically low on disk space.

Important: `compact` only removes fragmentation from MongoDB data files and does not return any disk space to the operating system.

To reclaim deleted space, use `repairDatabase`, which rebuilds the database which de-fragments the storage and may release space to the operating system. `repairDatabase` requires up to 2 gigabytes of extra disk space to run. Do not use `repairDatabase` if you are critically low on disk space.

Warning: `repairDatabase` requires enough free disk space to hold both the old and new database files while the repair is running. Be aware that `repairDatabase` will block all other operations and may take a long time to complete.

10.7.9 How can I check the size of a collection?

To view the size of a collection and other information, use the `db.collection.stats()` method from the `mongo` shell. The following example issues `db.collection.stats()` for the `orders` collection:

```
db.orders.stats();
```

To view specific measures of size, use these methods:

- `db.collection.dataSize()`: data size in bytes for the collection.
- `db.collection.storageSize()`: allocation size in bytes, including unused space.
- `db.collection.totalSize()`: the data size plus the index size in bytes.
- `db.collection.totalIndexSize()`: the index size in bytes.

Also, the following scripts print the statistics for each database and collection:

```
db._adminCommand("listDatabases").databases.forEach(function(d){mdb=db.getSiblingDB(d.name); print(mdb.stats());});  
db._adminCommand("listDatabases").databases.forEach(function(d){mdb=db.getSiblingDB(d.name); mdb.stats()});
```

10.7.10 How can I check the size of indexes?

To view the size of the data allocated for an index, use one of the following procedures in the `mongo` shell:

- Use the `db.collection.stats()` method using the index namespace. To retrieve a list of namespaces, issue the following command:

```
db.system.namespaces.find()
```

- Check the value of `indexSizes` in the output of the `db.collection.stats()` command.

Example

Issue the following command to retrieve index namespaces:

```
db.system.namespaces.find()
```

The command returns a list similar to the following:

```
{ "name" : "test.orders" }
{ "name" : "test.system.indexes" }
{ "name" : "test.orders.$_id_" }
```

View the size of the data allocated for the `orders.$_id_` index with the following sequence of operations:

```
use test
db.orders.$_id_.stats().indexSizes
```

10.7.11 How do I know when the server runs out of disk space?

If your server runs out of disk space for data files, you will see something like this in the log:

```
Thu Aug 11 13:06:09 [FileAllocator] allocating new data file dbms/test.13, filling with zeroes...
Thu Aug 11 13:06:09 [FileAllocator] error failed to allocate new file: dbms/test.13 size: 2146435072
Thu Aug 11 13:06:09 [FileAllocator]      will try again in 10 seconds
Thu Aug 11 13:06:19 [FileAllocator] allocating new data file dbms/test.13, filling with zeroes...
Thu Aug 11 13:06:19 [FileAllocator] error failed to allocate new file: dbms/test.13 size: 2146435072
Thu Aug 11 13:06:19 [FileAllocator]      will try again in 10 seconds
```

The server remains in this state forever, blocking all writes including deletes. However, reads still work. To delete some data and compact, using the `compact` command, you must restart the server first.

If your server runs out of disk space for journal files, the server process will exit. By default, `mongod` creates journal files in a sub-directory of `dbpath` named `journal`. You may elect to put the journal files on another storage device using a filesystem mount or a symlink.

Note: If you place the journal files on a separate storage device you will not be able to use a file system snapshot tool to capture a consistent snapshot of your data files and journal files.

10.8 FAQ: Indexes

Frequently Asked Questions:

- Should you run `ensureIndex()` after every insert? (page 604)
- How do you know what indexes exist in a collection? (page 604)
- How do you determine the size of an index? (page 604)
- What happens if an index does not fit into RAM? (page 604)
- How do you know what index a query used? (page 605)
- How do you determine what fields to index? (page 605)
- How do write operations affect indexes? (page 605)
- Will building a large index affect database performance? (page 605)
- Can I use index keys to constrain query matches? (page 605)
- Using `$ne` and `$nin` in a query is slow. Why? (page 605)
- Can I use a multi-key index to support a query for a whole array? (page 605)
- How can I effectively use indexes strategy for attribute lookups? (page 606)

This document addresses common questions regarding MongoDB indexes.

If you don't find the answer you're looking for, check the [complete list of FAQs](#) (page 571) or post your question to the MongoDB User Mailing List²⁶. See also [Indexing Tutorials](#) (page 334).

10.8.1 Should you run `ensureIndex()` after every insert?

No. You only need to create an index once for a single collection. After initial creation, MongoDB automatically updates the index as data changes.

While running `ensureIndex()` is usually ok, if an index doesn't exist because of ongoing administrative work, a call to `ensureIndex()` may disrupt database availability. Running `ensureIndex()` can render a replica set inaccessible as the index creation is happening. See [Build Indexes on Replica Sets](#) (page 339).

10.8.2 How do you know what indexes exist in a collection?

To list a collection's indexes, use the `db.collection.getIndexes()` method or a similar method for your driver²⁷.

10.8.3 How do you determine the size of an index?

To check the sizes of the indexes on a collection, use `db.collection.stats()`.

10.8.4 What happens if an index does not fit into RAM?

When an index is too large to fit into RAM, MongoDB must read the index from disk, which is a much slower operation than reading from RAM. Keep in mind an index fits into RAM when your server has RAM available for the index combined with the rest of the *working set*.

In certain cases, an index does not need to fit *entirely* into RAM. For details, see [Indexes that Hold Only Recent Values in RAM](#) (page 368).

²⁶<https://groups.google.com/forum/?fromgroups#!forum/mongodb-user>

²⁷<http://api.mongodb.org/>

10.8.5 How do you know what index a query used?

To inspect how MongoDB processes a query, use the `explain()` method in the `mongo` shell, or in your application driver.

10.8.6 How do you determine what fields to index?

A number of factors determine what fields to index, including [selectivity](#) (page 368), fitting indexes into RAM, reusing indexes in multiple queries when possible, and creating indexes that can support all the fields in a given query. For detailed documentation on choosing which fields to index, see [Indexing Tutorials](#) (page 334).

10.8.7 How do write operations affect indexes?

Any write operation that alters an indexed field requires an update to the index in addition to the document itself. If you update a document that causes the document to grow beyond the allotted record size, then MongoDB must update all indexes that include this document as part of the update operation.

Therefore, if your application is write-heavy, creating too many indexes might affect performance.

10.8.8 Will building a large index affect database performance?

Building an index can be an IO-intensive operation, especially if you have a large collection. This is true on any database system that supports secondary indexes, including MySQL. If you need to build an index on a large collection, consider building the index in the background. See [Index Creation](#) (page 332).

If you build a large index without the background option, and if doing so causes the database to stop responding, do one of the following:

- Wait for the index to finish building.
- Kill the current operation (see `db.killOp()`). The partial index will be deleted.

10.8.9 Can I use index keys to constrain query matches?

You can use the `min()` and `max()` methods to constrain the results of the cursor returned from `find()` by using index keys.

10.8.10 Using `$ne` and `$nin` in a query is slow. Why?

The `$ne` and `$nin` operators are not selective. See [Create Queries that Ensure Selectivity](#) (page 368). If you need to use these, it is often best to make sure that an additional, more selective criterion is part of the query.

10.8.11 Can I use a multi-key index to support a query for a whole array?

Not entirely. The index can partially support these queries because it can speed the selection of the first element of the array; however, comparing all subsequent items in the array cannot use the index and must scan the documents individually.

10.8.12 How can I effectively use indexes strategy for attribute lookups?

For simple attribute lookups that don't require sorted result sets or range queries, consider creating a field that contains an array of documents where each document has a field (e.g. `attrib`) that holds a specific type of attribute. You can index this `attrib` field.

For example, the `attrib` field in the following document allows you to add an unlimited number of attributes types:

```
{ _id : ObjectId(...),
  attrib : [
    { k: "color", v: "red" },
    { k: "shape": v: "rectangle" },
    { k: "color": v: "blue" },
    { k: "avail": v: true }
  ]
}
```

Both of the following queries could use the same `{ "attrib.k": 1, "attrib.v": 1 }` index:

```
db.mycollection.find( { attrib: { $elemMatch : { k: "color", v: "blue" } } } )
db.mycollection.find( { attrib: { $elemMatch : { k: "avail", v: true } } } )
```

10.9 FAQ: MongoDB Diagnostics

Frequently Asked Questions:

- Where can I find information about a `mongod` process that stopped running unexpectedly? (page 606)
- Does TCP keepalive time affect sharded clusters and replica sets? (page 607)
- What tools are available for monitoring MongoDB? (page 607)
- Memory Diagnostics (page 607)
 - Do I need to configure swap space? (page 607)
 - What is “working set” and how can I estimate its size? (page 608)
 - Must my working set size fit RAM? (page 608)
 - How do I calculate how much RAM I need for my application? (page 608)
 - How do I read memory statistics in the UNIX `top` command (page 608)
- Sharded Cluster Diagnostics (page 609)
 - In a new sharded cluster, why does all data remain on one shard? (page 609)
 - Why would one shard receive a disproportionate amount of traffic in a sharded cluster? (page 609)
 - What can prevent a sharded cluster from balancing? (page 609)
 - Why do chunk migrations affect sharded cluster performance? (page 610)

This document provides answers to common diagnostic questions and issues.

If you don't find the answer you're looking for, check the *complete list of FAQs* (page 571) or post your question to the MongoDB User Mailing List²⁸.

10.9.1 Where can I find information about a `mongod` process that stopped running unexpectedly?

If `mongod` shuts down unexpectedly on a UNIX or UNIX-based platform, and if `mongod` fails to log a shutdown or error message, then check your system logs for messages pertaining to MongoDB. For example, for logs located in

²⁸<https://groups.google.com/forum/?fromgroups#!forum/mongodb-user>

/var/log/messages, use the following commands:

```
sudo grep mongod /var/log/messages
sudo grep score /var/log/messages
```

10.9.2 Does TCP keepalive time affect sharded clusters and replica sets?

If you experience socket errors between members of a sharded cluster or replica set, that do not have other reasonable causes, check the TCP keep alive value, which Linux systems store as the `tcp_keepalive_time` value. A common keep alive period is 7200 seconds (2 hours); however, different distributions and OS X may have different settings. For MongoDB, you will have better experiences with shorter keepalive periods, on the order of 300 seconds (five minutes).

On Linux systems you can use the following operation to check the value of `tcp_keepalive_time`:

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

You can change the `tcp_keepalive_time` value with the following operation:

```
echo 300 > /proc/sys/net/ipv4/tcp_keepalive_time
```

The new `tcp_keepalive_time` value takes effect without requiring you to restart the `mongod` or `mongos` servers. When you reboot or restart your system you will need to set the new `tcp_keepalive_time` value, or see your operating system's documentation for setting the TCP keepalive value persistently.

For OS X systems, issue the following command to view the keep alive setting:

```
sysctl net.inet.tcp.keepinit
```

To set a shorter keep alive period use the following invocation:

```
sysctl -w net.inet.tcp.keepinit=300
```

If your replica set or sharded cluster experiences keepalive-related issues, you must alter the `tcp_keepalive_time` value on all machines hosting MongoDB processes. This includes all machines hosting `mongos` or `mongod` servers.

Windows users should consider the [Windows Server Technet Article on KeepAliveTime configuration²⁹](#) for more information on setting keep alive for MongoDB deployments on Windows systems.

10.9.3 What tools are available for monitoring MongoDB?

The *MongoDB Management Services* <<http://mms.mongodb.com>> includes monitoring. MMS Monitoring is a free, hosted services for monitoring MongoDB deployments. A full list of third-party tools is available as part of the [Monitoring for MongoDB](#) (page 136) documentation. Also consider the [MMS Documentation³⁰](#).

10.9.4 Memory Diagnostics

Do I need to configure swap space?

Always configure systems to have swap space. Without swap, your system may not be reliant in some situations with extreme memory constraints, memory leaks, or multiple programs using the same memory. Think of the swap space

²⁹http://technet.microsoft.com/en-us/library/dd349797.aspx#BKMK_2

³⁰<http://mms.mongodb.com/help/>

as something like a steam release valve that allows the system to release extra pressure without affecting the overall functioning of the system.

Nevertheless, systems running MongoDB *do not* need swap for routine operation. Database files are *memory-mapped* (page 600) and should constitute most of your MongoDB memory use. Therefore, it is unlikely that mongod will ever use any swap space in normal operation. The operating system will release memory from the memory mapped files without needing swap and MongoDB can write data to the data files without needing the swap system.

What is “working set” and how can I estimate its size?

The *working set* for a MongoDB database is the portion of your data that clients access most often. You can estimate size of the working set, using the `workingSet` document in the output of `serverStatus`. To return `serverStatus` with the `workingSet` document, issue a command in the following form:

```
db.runCommand( { serverStatus: 1, workingSet: 1 } )
```

Must my working set size fit RAM?

Your working set should stay in memory to achieve good performance. Otherwise many random disk IO’s will occur, and unless you are using SSD, this can be quite slow.

One area to watch specifically in managing the size of your working set is index access patterns. If you are inserting into indexes at random locations (as would happen with id’s that are randomly generated by hashes), you will continually be updating the whole index. If instead you are able to create your id’s in approximately ascending order (for example, day concatenated with a random id), all the updates will occur at the right side of the b-tree and the working set size for index pages will be much smaller.

It is fine if databases and thus virtual size are much larger than RAM.

How do I calculate how much RAM I need for my application?

The amount of RAM you need depends on several factors, including but not limited to:

- The relationship between *database storage* (page 599) and working set.
- The operating system’s cache strategy for LRU (Least Recently Used)
- The impact of *journaling* (page 229)
- The number or rate of page faults and other MMS gauges to detect when you need more RAM

MongoDB defers to the operating system when loading data into memory from disk. It simply *memory maps* (page 600) all its data files and relies on the operating system to cache data. The OS typically evicts the least-recently-used data from RAM when it runs low on memory. For example if clients access indexes more frequently than documents, then indexes will more likely stay in RAM, but it depends on your particular usage.

To calculate how much RAM you need, you must calculate your working set size, or the portion of your data that clients use most often. This depends on your access patterns, what indexes you have, and the size of your documents.

If page faults are infrequent, your working set fits in RAM. If fault rates rise higher than that, you risk performance degradation. This is less critical with SSD drives than with spinning disks.

How do I read memory statistics in the UNIX `top` command

Because mongod uses *memory-mapped files* (page 600), the memory statistics in `top` require interpretation in a special way. On a large database, `VSIZE` (virtual bytes) tends to be the size of the entire database. If the mongod

doesn't have other processes running, RSIZE (resident bytes) is the total memory of the machine, as this counts file system cache contents.

For Linux systems, use the `vmstat` command to help determine how the system uses memory. On OS X systems use `vm_stat`.

10.9.5 Sharded Cluster Diagnostics

The two most important factors in maintaining a successful sharded cluster are:

- *choosing an appropriate shard key* (page 499) and
- *sufficient capacity to support current and future operations* (page 497).

You can prevent most issues encountered with sharding by ensuring that you choose the best possible *shard key* for your deployment and ensure that you are always adding additional capacity to your cluster well before the current resources become saturated. Continue reading for specific issues you may encounter in a production environment.

In a new sharded cluster, why does all data remains on one shard?

Your cluster must have sufficient data for sharding to make sense. Sharding works by migrating chunks between the shards until each shard has roughly the same number of chunks.

The default chunk size is 64 megabytes. MongoDB will not begin migrations until the imbalance of chunks in the cluster exceeds the *migration threshold* (page 509). While the default chunk size is configurable with the `chunkSize` setting, these behaviors help prevent unnecessary chunk migrations, which can degrade the performance of your cluster as a whole.

If you have just deployed a sharded cluster, make sure that you have enough data to make sharding effective. If you do not have sufficient data to create more than eight 64 megabyte chunks, then all data will remain on one shard. Either lower the *chunk size* (page 511) setting, or add more data to the cluster.

As a related problem, the system will split chunks only on inserts or updates, which means that if you configure sharding and do not continue to issue insert and update operations, the database will not create any chunks. You can either wait until your application inserts data or *split chunks manually* (page 545).

Finally, if your shard key has a low *cardinality* (page 519), MongoDB may not be able to create sufficient splits among the data.

Why would one shard receive a disproportionate amount of traffic in a sharded cluster?

In some situations, a single shard or a subset of the cluster will receive a disproportionate portion of the traffic and workload. In almost all cases this is the result of a shard key that does not effectively allow *write scaling* (page 501).

It's also possible that you have "hot chunks." In this case, you may be able to solve the problem by splitting and then migrating parts of these chunks.

In the worst case, you may have to consider re-sharding your data and *choosing a different shard key* (page 518) to correct this pattern.

What can prevent a sharded cluster from balancing?

If you have just deployed your sharded cluster, you may want to consider the *troubleshooting suggestions for a new cluster where data remains on a single shard* (page 609).

If the cluster was initially balanced, but later developed an uneven distribution of data, consider the following possible causes:

- You have deleted or removed a significant amount of data from the cluster. If you have added additional data, it may have a different distribution with regards to its shard key.
- Your *shard key* has low [cardinality](#) (page 519) and MongoDB cannot split the chunks any further.
- Your data set is growing faster than the balancer can distribute data around the cluster. This is uncommon and typically is the result of:
 - a [balancing window](#) (page 539) that is too short, given the rate of data growth.
 - an uneven distribution of [write operations](#) (page 501) that requires more data migration. You may have to choose a different shard key to resolve this issue.
 - poor network connectivity between shards, which may lead to chunk migrations that take too long to complete. Investigate your network configuration and interconnections between shards.

Why do chunk migrations affect sharded cluster performance?

If migrations impact your cluster or application’s performance, consider the following options, depending on the nature of the impact:

1. If migrations only interrupt your clusters sporadically, you can limit the [balancing window](#) (page 539) to prevent balancing activity during peak hours. Ensure that there is enough time remaining to keep the data from becoming out of balance again.
2. If the balancer is always migrating chunks to the detriment of overall cluster performance:
 - You may want to attempt [decreasing the chunk size](#) (page 547) to limit the size of the migration.
 - Your cluster may be over capacity, and you may want to attempt to [add one or two shards](#) (page 521) to the cluster to distribute load.

It’s also possible that your shard key causes your application to direct all writes to a single shard. This kind of activity pattern can require the balancer to migrate most data soon after writing it. Consider redeploying your cluster with a shard key that provides better [write scaling](#) (page 501).

Release Notes

Always install the latest, stable version of MongoDB. See [MongoDB Version Numbers](#) (page 655) for more information.

See the following release notes for an account of the changes in major versions. Release notes also include instructions for upgrade.

11.1 Current Stable Release

(2.4-series)

11.1.1 Release Notes for MongoDB 2.4

March 19, 2013

Changes for 2.4

- Minor Releases (page 611)
- Major New Features (page 613)
- Security Enhancements (page 615)
- Performance Improvements (page 615)
- Enterprise (page 621)
- Additional Information (page 621)

MongoDB 2.4 includes enhanced geospatial support, switch to V8 JavaScript engine, security enhancements, and text search (beta) and hashed index.

Minor Releases

2.4.8 – November 1, 2013

- Increase future compatibility for 2.6 authorization features SERVER-11478¹.
- Fix dbhash cache issue for config servers SERVER-11421².

¹<https://jira.mongodb.org/browse/SERVER-11478>

²<https://jira.mongodb.org/browse/SERVER-11421>

- All 2.4.8 improvements³.

2.4.7 – October 21, 2013

- Fixed over-aggressive caching of V8 Isolates SERVER-10596⁴.
- Removed extraneous initial count during mapReduce SERVER-9907⁵.
- Cache results of dbhash command SERVER-11021⁶.
- Fixed memory leak in aggregation SERVER-10554⁷.
- All 2.4.7 improvements⁸.

2.4.6 – August 20, 2013

- Fix for possible loss of documents during the chunk migration process if a document in the chunk is very large SERVER-10478⁹.
- Fix for C++ client shutdown issues SERVER-8891¹⁰.
- Improved replication robustness in presence of high network latency SERVER-10085¹¹.
- Improved Solaris support SERVER-9832¹², SERVER-9786¹³, and SERVER-7080¹⁴.
- All 2.4.6 improvements¹⁵.

2.4.5 – July 3, 2013

- Fix for CVE-2013-4650 Improperly grant user system privileges on databases other than local SERVER-9983¹⁶.
- Fix for CVE-2013-3969 Remotely triggered segmentation fault in Javascript engine SERVER-9878¹⁷.
- Fix to prevent identical background indexes from being built SERVER-9856¹⁸.
- Config server performance improvements SERVER-9864¹⁹ and SERVER-5442²⁰.
- Improved initial sync resilience to network failure SERVER-9853²¹.
- All 2.4.5 improvements²².

³<https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.8%22%20AND%20project%20%3D%20SERVER>

⁴<https://jira.mongodb.org/browse/SERVER-10596>

⁵<https://jira.mongodb.org/browse/SERVER-9907>

⁶<https://jira.mongodb.org/browse/SERVER-11021>

⁷<https://jira.mongodb.org/browse/SERVER-10554>

⁸<https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.7%22%20AND%20project%20%3D%20SERVER>

⁹<https://jira.mongodb.org/browse/SERVER-10478>

¹⁰<https://jira.mongodb.org/browse/SERVER-8891>

¹¹<https://jira.mongodb.org/browse/SERVER-10085>

¹²<https://jira.mongodb.org/browse/SERVER-9832>

¹³<https://jira.mongodb.org/browse/SERVER-9786>

¹⁴<https://jira.mongodb.org/browse/SERVER-7080>

¹⁵<https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.6%22%20AND%20project%20%3D%20SERVER>

¹⁶<https://jira.mongodb.org/browse/SERVER-9983>

¹⁷<https://jira.mongodb.org/browse/SERVER-9878>

¹⁸<https://jira.mongodb.org/browse/SERVER-9856>

¹⁹<https://jira.mongodb.org/browse/SERVER-9864>

²⁰<https://jira.mongodb.org/browse/SERVER-5442>

²¹<https://jira.mongodb.org/browse/SERVER-9853>

²²<https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.5%22%20AND%20project%20%3D%20SERVER>

2.4.4 – June 4, 2013

- Performance fix for Windows version SERVER-9721²³
- Fix for config upgrade failure SERVER-9661²⁴.
- Migration to Cyrus SASL library for MongoDB Enterprise SERVER-8813²⁵.
- All 2.4.4 improvements²⁶.

2.4.3 – April 23, 2013

- Fix for mongo shell ignoring modified object's _id field SERVER-9385²⁷.
- Fix for race condition in log rotation SERVER-4739²⁸.
- Fix for copydb command with authorization in a sharded cluster SERVER-9093²⁹.
- All 2.4.3 improvements³⁰.

2.4.2 – April 17, 2013

- Several V8 memory leak and performance fixes SERVER-9267³¹ and SERVER-9230³².
- Fix for upgrading sharded clusters SERVER-9125³³.
- Fix for high volume connection crash SERVER-9014³⁴.
- All 2.4.2 improvements³⁵

2.4.1 – April 17, 2013

- Fix for losing index changes during initial sync SERVER-9087³⁶
- All 2.4.1 improvements³⁷.

Major New Features

The following changes in MongoDB affect both standard and Enterprise editions:

²³<https://jira.mongodb.org/browse/SERVER-9721>

²⁴<https://jira.mongodb.org/browse/SERVER-9661>

²⁵<https://jira.mongodb.org/browse/SERVER-8813>

²⁶<https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.4%22%20AND%20project%20%3D%20SERVER>

²⁷<https://jira.mongodb.org/browse/SERVER-9385>

²⁸<https://jira.mongodb.org/browse/SERVER-4739>

²⁹<https://jira.mongodb.org/browse/SERVER-9093>

³⁰<https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.3%22%20AND%20project%20%3D%20SERVER>

³¹<https://jira.mongodb.org/browse/SERVER-9267>

³²<https://jira.mongodb.org/browse/SERVER-9230>

³³<https://jira.mongodb.org/browse/SERVER-9125>

³⁴<https://jira.mongodb.org/browse/SERVER-9014>

³⁵<https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.2%22%20AND%20project%20%3D%20SERVER>

³⁶<https://jira.mongodb.org/browse/SERVER-9087>

³⁷<https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.1%22%20AND%20project%20%3D%20SERVER>

Text Search

Add support for text search of content in MongoDB databases as a *beta* feature. See [Text Indexes](#) (page 328) for more information.

Geospatial Support Enhancements

- Add new [2dsphere index](#) (page 324). The new index supports GeoJSON³⁸ objects Point, LineString, and Polygon. See [2dsphere Indexes](#) (page 324) and [Geospatial Indexes and Queries](#) (page 322).
- Introduce operators \$geometry, \$geoWithin and \$geoIntersects to work with the GeoJSON data.

Hashed Index

Add new [hashed index](#) (page 329) to index documents using hashes of field values. When used to index a shard key, the hashed index ensures an evenly distributed shard key. See also [Hashed Shard Keys](#) (page 500).

Improvements to the Aggregation Framework

- Improve support for geospatial queries. See the \$geoWithin operator and the \$geoNear pipeline stage.
- Improve sort efficiency when the \$sort stage immediately precedes a \$limit in the pipeline.
- Add new operators \$millisecond and \$concat and modify how \$min operator processes null values.

Changes to Update Operators

- Add new \$setOnInsert operator for use with upsert .
- Enhance functionality of the \$push operator, supporting its use with the \$each, the \$sort, and the \$slice modifiers.

Additional Limitations for Map-Reduce and \$where Operations

The mapReduce command, group command, and the \$where operator expressions cannot access certain global functions or properties, such as db, that are available in the mongo shell. See the individual command or operator for details.

Improvements to serverStatus Command

Provide additional metrics and customization for the serverStatus command. See `db.serverStatus()` and `serverStatus` for more information.

³⁸<http://geojson.org/geojson-spec.html>

Security Enhancements

- Introduce a role-based access control system [User Privilege Roles in MongoDB](#) (page 263) using new `system.users Privilege Documents` (page 268).
- Enforce uniqueness of the user in user privilege documents per database. Previous versions of MongoDB did not enforce this requirement, and existing databases may have duplicates.
- Support encrypted connections using SSL certificates signed by a Certificate Authority. See [Connect to MongoDB with SSL](#) (page 248).

For more information on security and risk management strategies, see [MongoDB Security Practices and Procedures](#) (page 233).

Performance Improvements

V8 JavaScript Engine

JavaScript Changes in MongoDB 2.4 Consider the following impacts of [V8 JavaScript Engine](#) (page 615) in MongoDB 2.4:

Tip

Use the new `interpreterVersion()` method in the `mongo` shell and the `javascriptEngine` field in the output of `db.serverBuildInfo()` to determine which JavaScript engine a MongoDB binary uses.

Improved Concurrency Previously, 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. The switch to V8 improves concurrency by permitting multiple JavaScript operations to run at the same time.

Modernized JavaScript Implementation (ES5) The 5th edition of [ECMAScript³⁹](#), abbreviated as ES5, adds many new language features, including:

- standardized [JSON⁴⁰](#),
- [strict mode⁴¹](#),
- [function.bind\(\)⁴²](#),
- [array extensions⁴³](#), and
- getters and setters.

With V8, MongoDB supports the ES5 implementation of Javascript with the following exceptions.

Note: The following features do not work as expected on documents **returned from MongoDB queries**:

- `Object.seal()` throws an exception on documents returned from MongoDB queries.
- `Object.freeze()` throws an exception on documents returned from MongoDB queries.
- `Object.preventExtensions()` incorrectly allows the addition of new properties on documents returned from MongoDB queries.

³⁹<http://www.ecma-international.org/publications/standards/Ecma-262.htm>

⁴⁰<http://www.ecma-international.org/ecma-262/5.1/#sec-15.12.1>

⁴¹<http://www.ecma-international.org/ecma-262/5.1/#sec-4.2.2>

⁴²<http://www.ecma-international.org/ecma-262/5.1/#sec-15.3.4.5>

⁴³<http://www.ecma-international.org/ecma-262/5.1/#sec-15.4.4.16>

- enumerable properties, when added to documents returned from MongoDB queries, are not saved during write operations.

See SERVER-8216⁴⁴, SERVER-8223⁴⁵, SERVER-8215⁴⁶, and SERVER-8214⁴⁷ for more information.

For objects that have not been returned from MongoDB queries, the features work as expected.

Removed Non-Standard SpiderMonkey Features V8 does **not** support the following *non-standard* SpiderMonkey⁴⁸ JavaScript extensions, previously supported by MongoDB's use of SpiderMonkey as its JavaScript engine.

E4X Extensions V8 does not support the *non-standard* E4X⁴⁹ extensions. E4X provides a native XML⁵⁰ object to the JavaScript language and adds the syntax for embedding literal XML documents in JavaScript code.

You need to use alternative XML processing if you used any of the following constructors/methods:

- XML ()
- Namespace ()
- QName ()
- XMLElement ()
- isXMLName ()

Destructuring Assignment V8 does not support the non-standard destructuring assignments. Destructuring assignment “extract[s] data from arrays or objects using a syntax that mirrors the construction of array and object literals.” - Mozilla docs⁵¹

Example

The following destructuring assignment is **invalid** with V8 and throws a SyntaxError:

```
original = [4, 8, 15];
var [b, ,c] = a; // <== destructuring assignment
print(b) // 4
print(c) // 15
```

Iterator(), StopIteration(), and Generators V8 does not support Iterator(), StopIteration(), and generators⁵².

InternalError() V8 does not support InternalError(). Use Error() instead.

⁴⁴<https://jira.mongodb.org/browse/SERVER-8216>

⁴⁵<https://jira.mongodb.org/browse/SERVER-8223>

⁴⁶<https://jira.mongodb.org/browse/SERVER-8215>

⁴⁷<https://jira.mongodb.org/browse/SERVER-8214>

⁴⁸<https://developer.mozilla.org/en-US/docs/SpiderMonkey>

⁴⁹<https://developer.mozilla.org/en-US/docs/E4X>

⁵⁰https://developer.mozilla.org/en-US/docs/E4X/Processing_XML_with_E4X

⁵¹[https://developer.mozilla.org/en-US/docs/JavaScript/New_in_JavaScript/1.7#Destructuring_assignment_\(Merge_into_own_page.2Fsection\)](https://developer.mozilla.org/en-US/docs/JavaScript/New_in_JavaScript/1.7#Destructuring_assignment_(Merge_into_own_page.2Fsection))

⁵²https://developer.mozilla.org/en-US/docs/JavaScript/Guide/Iterators_and_Generators

for each...in Construct V8 does not support the use of `for each...in`⁵³ construct. Use `for (var x in y)` construct instead.

Example

The following `for each (var x in y)` construct is **invalid** with V8:

```
var o = { name: 'MongoDB', version: 2.4 };

for each (var value in o) {
  print(value);
}
```

Instead, in version 2.4, you can use the `for (var x in y)` construct:

```
var o = { name: 'MongoDB', version: 2.4 };

for (var prop in o) {
  var value = o[prop];
  print(value);
}
```

You can also use the array *instance* method `forEach()` with the ES5 method `Object.keys()`:

```
Object.keys(o).forEach(function (key) {
  var value = o[key];
  print(value);
});
```

Array Comprehension V8 does not support `Array comprehensions`⁵⁴.

Use other methods such as the `Array` instance methods `map()`, `filter()`, or `forEach()`.

Example

With V8, the following array comprehension is **invalid**:

```
var a = { w: 1, x: 2, y: 3, z: 4 }

var arr = [i * i for each (i in a) if (i > 2)]
printjson(arr)
```

Instead, you can implement using the `Array` *instance* method `forEach()` and the ES5 method `Object.keys()`:

```
var a = { w: 1, x: 2, y: 3, z: 4 }

var arr = [];
Object.keys(a).forEach(function (key) {
  var val = a[key];
  if (val > 2) arr.push(val * val);
})
printjson(arr)
```

Note: The new logic uses the `Array` *instance* method `forEach()` and not the *generic* method

⁵³https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Statements/for_each...in

⁵⁴https://developer.mozilla.org/en-US/docs/JavaScript/Guide/Predefined_Core_Objects#Array_comprehensions

`Array.forEach()`; V8 does **not** support `Array generic` methods. See *Array Generic Methods* (page 619) for more information.

Multiple Catch Blocks V8 does not support multiple `catch` blocks and will throw a `SyntaxError`.

Example

The following multiple catch blocks is **invalid** with V8 and will throw "SyntaxError: Unexpected token if":

```
try {
    something()
} catch (err if err instanceof SomeError) {
    print('some error')
} catch (err) {
    print('standard error')
}
```

Conditional Function Definition V8 will produce different outcomes than SpiderMonkey with conditional function definitions⁵⁵.

Example

The following conditional function definition produces different outcomes in SpiderMonkey versus V8:

```
function test () {
    if (false) {
        function go () {};
    }
    print(typeof go)
}
```

With SpiderMonkey, the conditional function outputs `undefined`, whereas with V8, the conditional function outputs `function`.

If your code defines functions this way, it is highly recommended that you refactor the code. The following example refactors the conditional function definition to work in both SpiderMonkey and V8.

```
function test () {
    var go;
    if (false) {
        go = function () {}
    }
    print(typeof go)
}
```

The refactored code outputs `undefined` in both SpiderMonkey and V8.

Note: ECMAScript prohibits conditional function definitions. To force V8 to throw an Error, enable strict mode⁵⁶.

```
function test () {
    'use strict';
```

⁵⁵<https://developer.mozilla.org/en-US/docs/JavaScript/Guide/Functions>

⁵⁶<http://www.nczonline.net/blog/2012/03/13/its-time-to-start-using-javascript-strict-mode/>

```

if (false) {
  function go () {}
}
}

```

The JavaScript code throws the following syntax error:

```
SyntaxError: In strict mode code, functions can only be declared at top level or immediately within a
```

String Generic Methods V8 does not support [String generics](#)⁵⁷. String generics are a set of methods on the `String` class that mirror instance methods.

Example

The following use of the generic method `String.toLowerCase()` is **invalid** with V8:

```

var name = 'MongoDB';

var lower = String.toLowerCase(name);

```

With V8, use the `String` instance method `toLowerCase()` available through an *instance* of the `String` class instead:

```

var name = 'MongoDB';

var lower = name.toLowerCase();
print(name + ' becomes ' + lower);

```

With V8, use the `String` *instance* methods instead of following *generic* methods:

<code>String.charAt()</code>	<code>String.quote()</code>	<code>String.toLocaleLowerCase()</code>
<code>String.charCodeAt()</code>	<code>String.replace()</code>	<code>String.toLocaleUpperCase()</code>
<code>String.concat()</code>	<code>String.search()</code>	<code>String.toLowerCase()</code>
<code>String.endsWith()</code>	<code>String.slice()</code>	<code>String.toUpperCase()</code>
<code>String.indexOf()</code>	<code>String.split()</code>	<code>String.trim()</code>
<code>String.lastIndexOf()</code>	<code>String.startsWith()</code>	<code>String.trimLeft()</code>
<code>String.localeCompare()</code>	<code>String.substr()</code>	<code>String.trimRight()</code>
<code>String.match()</code>	<code>String.substring()</code>	

Array Generic Methods V8 does not support [Array generic methods](#)⁵⁸. Array generics are a set of methods on the `Array` class that mirror instance methods.

Example

The following use of the generic method `Array.every()` is **invalid** with V8:

```

var arr = [4, 8, 15, 16, 23, 42];

function isEven (val) {
  return 0 === val % 2;
}

```

⁵⁷https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/String#String_generic_methods

⁵⁸https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/Array#Array_generic_methods

```
var allEven = Array.every(arr, isEven);
print(allEven);
```

With V8, use the `Array` instance method `every()` available through an *instance* of the `Array` class instead:

```
var allEven = arr.every(isEven);
print(allEven);
```

With V8, use the `Array` *instance* methods instead of the following *generic* methods:

<code>Array.concat()</code>	<code>Array.lastIndexOf()</code>	<code>Array.slice()</code>
<code>Array.every()</code>	<code>Array.map()</code>	<code>Array.some()</code>
<code>Array.filter()</code>	<code>Array.pop()</code>	<code>Array.sort()</code>
<code>Array.forEach()</code>	<code>Array.push()</code>	<code>Array.splice()</code>
<code>Array.indexOf()</code>	<code>Array.reverse()</code>	<code>Array.unshift()</code>
<code>Array.join()</code>	<code>Array.shift()</code>	

Array Instance Method `toSource()` V8 does not support the `Array` instance method `toSource()`⁵⁹. Use the `Array` instance method `toString()` instead.

`uneval()` V8 does not support the non-standard method `uneval()`. Use the standardized `JSON.stringify()`⁶⁰ method instead.

Change default JavaScript engine from SpiderMonkey to V8. The change provides improved concurrency for JavaScript operations, modernized JavaScript implementation, and the removal of non-standard SpiderMonkey features, and affects all JavaScript behavior including the commands `mapReduce`, `group`, and `eval` and the query operator `$where`.

See [JavaScript Changes in MongoDB 2.4](#) (page 615) for more information about all changes .

BSON Document Validation Enabled by Default for `mongod` and `mongorestore`

Enable basic `BSON` object validation for `mongod` and `mongorestore` when writing to MongoDB data files. See `objcheck` for details.

Index Build Enhancements

- Add support for multiple concurrent index builds in the background by a single `mongod` instance. See [building indexes in the background](#) (page 332) for more information on background index builds.
- Allow the `db.killOp()` method to terminate a foreground index build.
- Improve index validation during index creation. See [Compatibility and Index Type Changes in MongoDB 2.4](#) (page 628) for more information.

Set Parameters as Command Line Options

Provide `--setParameter` as a command line option for `mongos` and `mongod`. See `mongod` and `mongos` for list of available options for `setParameter`.

⁵⁹https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/Array/toSource

⁶⁰https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/JSON/stringify

Increased Chunk Migration Write Concern

The default write concern for insert and delete operations that occur as part of a *chunk* migration in a *sharded cluster* now ensures that at least one secondary acknowledges each insert and deletion operation. See [Chunk Migration Write Concern](#) (page 511).

Improved Chunk Migration Queue Behavior

Increase performance for moving multiple chunks off an overloaded shard. The balancer no longer waits for the current migration's delete phase to complete before starting the next chunk migration. See [Chunk Migration Queuing](#) (page 511) for details.

Enterprise

The following changes are specific to MongoDB Enterprise Editions:

SASL Library Change

In 2.4.4, MongoDB Enterprise uses Cyrus SASL. Earlier 2.4 Enterprise versions use GNU SASL (libgsasl). To upgrade to 2.4.4 MongoDB Enterprise or greater, you **must** install all package dependencies related to this change, including the appropriate Cyrus SASL GSSAPI library. See [Install MongoDB Enterprise](#) (page 15) for details of the dependencies.

New Modular Authentication System with Support for Kerberos

In 2.4, the MongoDB Enterprise now supports authentication via a Kerberos mechanism. See [Deploy MongoDB with Kerberos Authentication](#) (page 257) for more information. For drivers that provide support for Kerberos authentication to MongoDB, refer to [Use MongoDB Drivers to Authenticate with Kerberos](#) (page 260).

For more information on security and risk management strategies, see [MongoDB Security Practices and Procedures](#) (page 233).

Additional Information

Platform Notes

For OS X, MongoDB 2.4 only supports OS X versions 10.6 (Snow Leopard) and later. There are no other platform support changes in MongoDB 2.4. See the [downloads page](#)⁶¹ for more information on platform support.

Upgrade Process

Upgrade MongoDB to 2.4 In the general case, the upgrade from MongoDB 2.2 to 2.4 is a binary-compatible “drop-in” upgrade: shut down the mongod instances and replace them with mongod instances running 2.4. **However**, before you attempt any upgrade please familiarize yourself with the content of this document, particularly the procedure for [upgrading sharded clusters](#) (page 623) and the considerations for [reverting to 2.2 after running 2.4](#) (page 626).

⁶¹<http://www.mongodb.org/downloads/>

Content

- Upgrade Recommendations and Checklist (page 622)
- Upgrade Standalone mongod Instance to MongoDB 2.4 (page 622)
- Upgrade a Replica Set from MongoDB 2.2 to MongoDB 2.4 (page 622)
- Upgrade a Sharded Cluster from MongoDB 2.2 to MongoDB 2.4 (page 623)
- Rolling Upgrade Limitation for 2.2.0 Deployments Running with auth Enabled (page 626)
- Upgrade from 2.3 to 2.4 (page 626)
- Downgrade MongoDB from 2.4 to Previous Versions (page 626)

Upgrade Recommendations and Checklist When upgrading, consider the following:

- For all deployments using authentication, upgrade the drivers (i.e. client libraries), before upgrading the mongod instance or instances.
- To upgrade to 2.4 sharded clusters *must* upgrade following the *meta-data upgrade procedure* (page 623).
- If you’re using 2.2.0 and running with auth enabled, you will need to upgrade first to 2.2.1 and then upgrade to 2.4. See *Rolling Upgrade Limitation for 2.2.0 Deployments Running with auth Enabled* (page 626).
- If you have `system.users` (page 268) documents (i.e. for auth) that you created before 2.4 you *must* ensure that there are no duplicate values for the `user` field in the `system.users` (page 268) collection in *any* database. If you *do* have documents with duplicate user fields, you must remove them before upgrading.

See *Security Enhancements* (page 615) for more information.

Upgrade Standalone mongod Instance to MongoDB 2.4

1. Download binaries of the latest release in the 2.4 series from the [MongoDB Download Page](#)⁶². See *Install MongoDB* (page 3) for more information.
2. Shutdown your mongod instance. Replace the existing binary with the 2.4 mongod binary and restart mongod.

Upgrade a Replica Set from MongoDB 2.2 to MongoDB 2.4 You can upgrade to 2.4 by performing a “rolling” upgrade of the set by upgrading the members individually while the other members are available to minimize downtime. Use the following procedure:

1. Upgrade the *secondary* members of the set one at a time by shutting down the mongod and replacing the 2.2 binary with the 2.4 binary. After upgrading a mongod instance, wait for the member to recover to SECONDARY state before upgrading the next instance. To check the member’s state, issue `rs.status()` in the mongo shell.
2. Use the mongo shell method `rs.stepDown()` to step down the *primary* to allow the normal *failover* (page 392) procedure. `rs.stepDown()` expedites the failover procedure and is preferable to shutting down the primary directly.

Once the primary has stepped down and another member has assumed PRIMARY state, as observed in the output of `rs.status()`, shut down the previous primary and replace mongod binary with the 2.4 binary and start the new process.

Note: Replica set failover is not instant but will render the set unavailable to read or accept writes until the failover process completes. Typically this takes 10 seconds or more. You may wish to plan the upgrade during a predefined maintenance window.

⁶²<http://www.mongodb.org/downloads>

Upgrade a Sharded Cluster from MongoDB 2.2 to MongoDB 2.4

Important: Only upgrade sharded clusters to 2.4 if **all** members of the cluster are currently running instances of 2.2. The only supported upgrade path for sharded clusters running 2.0 is via 2.2.

Upgrading a *sharded cluster* from MongoDB version 2.2 to 2.4 (or 2.3) requires that you run a 2.4 mongos with the `--upgrade` option, described in this procedure. The upgrade process does not require downtime.

The upgrade to MongoDB 2.4 adds epochs to the meta-data for all collections and chunks in the existing cluster. MongoDB 2.2 processes are capable of handling epochs, even though 2.2 did not require them.

This procedure applies only to upgrades from version 2.2. Earlier versions of MongoDB do not correctly handle epochs.

Warning:

- Before you start the upgrade, ensure that the amount of free space on the filesystem for the *config database* (page 555) is 4 to 5 times the amount of space currently used by the *config database* (page 555) data files. Additionally, ensure that all indexes in the *config database* (page 555) are `{v:1}` indexes. If a critical index is a `{v:0}` index, chunk splits can fail due to known issues with the `{v:0}` format. `{v:0}` indexes are present on clusters created with MongoDB 2.0 or earlier.
The duration of the metadata upgrade depends on the network latency between the node performing the upgrade and the three config servers. Ensure low latency between the upgrade process and the config servers.
- While the upgrade is in progress, you cannot make changes to the collection meta-data. For example, during the upgrade, do **not** perform:
 - `sh.enableSharding()`,
 - `sh.shardCollection()`,
 - `sh.addShard()`,
 - `db.createCollection()`,
 - `db.collection.drop()`,
 - `db.dropDatabase()`,
 - any operation that creates a database, or
 - any other operation that modifies the cluster meta-data in any way. See *Sharding Reference* (page 554) for a complete list of sharding commands. Note, however, that not all commands on the *Sharding Reference* (page 554) page modifies the cluster meta-data.
- Once you upgrade to 2.4 and complete the upgrade procedure **do not** use 2.0 mongod and mongos processes in your cluster. 2.0 process may re-introduce old meta-data formats into cluster meta-data.

Note: The upgraded config database will require more storage space than before, to make backup and working copies of the `config.chunks` (page 557) and `config.collections` (page 558) collections. As always, if storage requirements increase, the mongod might need to pre-allocate additional data files. See *What tools can I use to investigate storage use in MongoDB?* (page 601) for more information.

Meta-Data Upgrade Procedure Changes to the meta-data format for sharded clusters, stored in the *config database* (page 555), require a special meta-data upgrade procedure when moving to 2.4.

Do not perform operations that modify meta-data while performing this procedure. See *Upgrade a Sharded Cluster from MongoDB 2.2 to MongoDB 2.4* (page 623) for examples of prohibited operations.

1. Before you start the upgrade, ensure that the amount of free space on the filesystem for the *config database* (page 555) is 4 to 5 times the amount of space currently used by the *config database* (page 555) data files. Additionally, ensure that all indexes in the *config database* (page 555) are `{v:1}` indexes. If a critical index is a `{v:0}` index, chunk splits can fail due to known issues with the `{v:0}` format. `{v:0}` indexes are present on clusters created with MongoDB 2.0 or earlier.

The duration of the metadata upgrade depends on the network latency between the node performing the upgrade and the three config servers. Ensure low latency between the upgrade process and the config servers.

To check the version of your indexes, use `db.collection.getIndexes()`.

If any index **on the config database** is `{v:0}`, you should rebuild those indexes by connecting to the mongos and either: rebuild all indexes using the `db.collection.reIndex()` method, or drop and rebuild specific indexes using `db.collection.dropIndex()` and then `db.collection.ensureIndex()`. If you need to upgrade the `_id` index to `{v:1}` use `db.collection.reIndex()`.

You may have `{v:0}` indexes on other databases in the cluster.

2. Turn off the [balancer](#) (page 508) in the *sharded cluster*, as described in [Disable the Balancer](#) (page 540).
-

Optional

For additional security during the upgrade, you can make a backup of the config database using `mongodump` or other backup tools.

3. Ensure there are no version 2.0 mongod or mongos processes still active in the sharded cluster. The automated upgrade process checks for 2.0 processes, but network availability can prevent a definitive check. Wait 5 minutes after stopping or upgrading version 2.0 mongos processes to confirm that none are still active.
4. Start a single 2.4 mongos process with `configdb` pointing to the sharded cluster's [config servers](#) (page 496) and with the `--upgrade` option. The upgrade process happens before the process becomes a daemon (i.e. before `--fork`.)

You can upgrade an existing mongos instance to 2.4 or you can start a new *mongos* instance that can reach all config servers if you need to avoid reconfiguring a production mongos.

Start the mongos with a command that resembles the following:

```
mongos --configdb <config servers> --upgrade
```

Without the `--upgrade` option 2.4 mongos processes will fail to start until the upgrade process is complete.

The upgrade will prevent any chunk moves or splits from occurring during the upgrade process. If there are very many sharded collections or there are stale locks held by other failed processes, acquiring the locks for all collections can take seconds or minutes. See the log for progress updates.

5. When the mongos process starts successfully, the upgrade is complete. If the mongos process fails to start, check the log for more information.

If the mongos terminates or loses its connection to the config servers during the upgrade, you may always safely retry the upgrade.

However, if the upgrade failed during the short critical section, the mongos will exit and report that the upgrade will require manual intervention. To continue the upgrade process, you must follow the [Resync after an Interruption of the Critical Section](#) (page 625) procedure.

Optional

If the mongos logs show the upgrade waiting for the upgrade lock, a previous upgrade process may still be active or may have ended abnormally. After 15 minutes of no remote activity mongos will force the upgrade lock. If you can verify that there are no running upgrade processes, you may connect to a 2.2 mongos process and force the lock manually:

```
mongo <mongos.example.net>
```

```
db.getMongo().getCollection("config.locks").findOne({ _id : "configUpgrade" })
```

If the process specified in the `process` field of this document is *verifiably* offline, run the following operation to force the lock.

```
db.getMongo().getCollection("config.locks").update({ _id : "configUpgrade" }, { $set : { state :
```

It is always more safe to wait for the mongos to verify that the lock is inactive, if you have any doubts about the activity of another upgrade operation. In addition to the `configUpgrade`, the mongos may need to wait for specific collection locks. Do not force the specific collection locks.

-
6. Upgrade and restart other mongos processes in the sharded cluster, *without* the `--upgrade` option.

See [Complete Sharded Cluster Upgrade](#) (page 626) for more information.

7. [Re-enable the balancer](#) (page 540). You can now perform operations that modify cluster meta-data.

Once you have upgraded, *do not* introduce version 2.0 MongoDB processes into the sharded cluster. This can reintroduce old meta-data formats into the config servers. The meta-data change made by this upgrade process will help prevent errors caused by cross-version incompatibilities in future versions of MongoDB.

Resync after an Interruption of the Critical Section During the short critical section of the upgrade that applies changes to the meta-data, it is unlikely but possible that a network interruption can prevent all three config servers from verifying or modifying data. If this occurs, the [config servers](#) (page 496) must be re-synced, and there may be problems starting new mongos processes. The *sharded cluster* will remain accessible, but avoid all cluster meta-data changes until you resync the config servers. Operations that change meta-data include: adding shards, dropping databases, and dropping collections.

Note: Only perform the following procedure *if* something (e.g. network, power, etc.) interrupts the upgrade process during the short critical section of the upgrade. Remember, you may always safely attempt the [meta data upgrade procedure](#) (page 623).

To resync the config servers:

1. Turn off the [balancer](#) (page 508) in the sharded cluster and stop all meta-data operations. If you are in the middle of an upgrade process ([Upgrade a Sharded Cluster from MongoDB 2.2 to MongoDB 2.4](#) (page 623)), you have already disabled the balancer.
2. Shut down two of the three config servers, preferably the last two listed in the `configdb` string. For example, if your `configdb` string is `configA:27019,configB:27019,configC:27019`, shut down `configB` and `configC`. Shutting down the last two config servers ensures that most mongos instances will have uninterrupted access to cluster meta-data.
3. `mongodump` the data files of the active config server (`configA`).
4. Move the data files of the deactivated config servers (`configB` and `configC`) to a backup location.
5. Create new, empty *data directories*.
6. Restart the disabled config servers with `--dbpath` pointing to the now-empty data directory and `--port` pointing to an alternate port (e.g. 27020).
7. Use `mongorestore` to repopulate the data files on the disabled documents from the active config server (`configA`) to the restarted config servers on the new port (`configB:27020,configC:27020`). These config servers are now re-synced.
8. Restart the restored config servers on the old port, resetting the port back to the old settings (`configB:27019` and `configC:27019`).
9. In some cases connection pooling may cause spurious failures, as the mongos disables old connections only after attempted use. 2.4 fixes this problem, but to avoid this issue in version 2.2, you can restart all mongos

instances (one-by-one, to avoid downtime) and use the `rs.stepDown()` method before restarting each of the shard *replica set primaries*.

10. The sharded cluster is now fully resynced; however before you attempt the upgrade process again, you must manually reset the upgrade state using a version 2.2 mongos. Begin by connecting to the 2.2 mongos with the mongo shell:

```
mongo <mongos.example.net>
```

Then, use the following operation to reset the upgrade process:

```
db.getMongo().getCollection("config.version").update({ _id : 1 }, { $unset : { upgradeState : 1 }}
```

11. Finally retry the upgrade process, as in [Upgrade a Sharded Cluster from MongoDB 2.2 to MongoDB 2.4](#) (page 623).

Complete Sharded Cluster Upgrade After you have successfully completed the meta-data upgrade process described in [Meta-Data Upgrade Procedure](#) (page 623), and the 2.4 mongos instance starts, you can upgrade the other processes in your MongoDB deployment.

While the balancer is still disabled, upgrade the components of your sharded cluster in the following order:

- Upgrade all mongos instances in the cluster, in any order.
- Upgrade all 3 mongod config server instances, upgrading the *first* system in the `mongos --configdb` argument *last*.
- Upgrade each shard, one at a time, upgrading the mongod secondaries before running `replSetStepDown` and upgrading the primary of each shard.

When this process is complete, you can now [re-enable the balancer](#) (page 540).

Rolling Upgrade Limitation for 2.2.0 Deployments Running with auth Enabled MongoDB *cannot* support deployments that mix 2.2.0 and 2.4.0, or greater, components. MongoDB version 2.2.1 and later processes *can* exist in mixed deployments with 2.4-series processes. Therefore you cannot perform a rolling upgrade from MongoDB 2.2.0 to MongoDB 2.4.0. To upgrade a cluster with 2.2.0 components, use one of the following procedures.

1. Perform a rolling upgrade of all 2.2.0 processes to the latest 2.2-series release (e.g. 2.2.3) so that there are no processes in the deployment that predate 2.2.1. When there are no 2.2.0 processes in the deployment, perform a rolling upgrade to 2.4.0.
2. Stop all processes in the cluster. Upgrade all processes to a 2.4-series release of MongoDB, and start all processes at the same time.

Upgrade from 2.3 to 2.4 If you used a mongod from the 2.3 or 2.4-rc (release candidate) series, you can safely transition these databases to 2.4.0 or later; *however*, if you created 2dsphere or text indexes using a mongod before v2.4-rc2, you will need to rebuild these indexes. For example:

```
db.records.dropIndex( { loc: "2dsphere" } )
db.records.dropIndex( "records_text" )

db.records.ensureIndex( { loc: "2dsphere" } )
db.records.ensureIndex( { records: "text" } )
```

Downgrade MongoDB from 2.4 to Previous Versions For some cases the on-disk format of data files used by 2.4 and 2.2 mongod is compatible, and you can upgrade and downgrade if needed. However, several new features in 2.4 are incompatible with previous versions:

- 2dsphere indexes are incompatible with 2.2 and earlier mongod instances.
- text indexes are incompatible with 2.2 and earlier mongod instances.
- using a hashed index as a shard key are incompatible with 2.2 and earlier mongos instances.
- hashed indexes are incompatible with 2.0 and earlier mongod instances.

Important: Collections sharded using hashed shard keys, should **not** use 2.2 mongod instances, which cannot correctly support cluster operations for these collections.

If you completed the [meta-data upgrade for a sharded cluster](#) (page 623), you can safely downgrade to 2.2 MongoDB processes. **Do not** use 2.0 processes after completing the upgrade procedure.

Note: In sharded clusters, once you have completed the [meta-data upgrade procedure](#) (page 623), you cannot use 2.0 mongod or mongos instances in the same cluster.

If you complete the meta-data upgrade, you can have a mixed cluster that has both 2.2 and 2.4 mongod and mongos instances, if needed. However, **do not** create 2dsphere or text indexes in a cluster that has 2.2 components.

Considerations and Compatibility If you upgrade to MongoDB 2.4, and then need to run MongoDB 2.2 with the same data files, consider the following limitations.

- If you use a hashed index as the shard key index, which is only possible under 2.4 you will not be able to query data in this sharded collection. Furthermore, a 2.2 mongos cannot properly route an insert operation for a collections sharded using a hashed index for the shard key index: any data that you insert using a 2.2 mongos, will not arrive on the correct shard and will not be reachable by future queries.
- If you *never* create an 2dsphere or text index, you can move between a 2.4 and 2.2 mongod for a given data set; however, after you create the first 2dsphere or text index with a 2.4 mongod you will need to run a 2.2 mongod with the `--upgrade` option and drop any 2dsphere or text index.

Upgrade and Downgrade Procedures

Basic Downgrade and Upgrade Except as described below, moving between 2.2 and 2.4 is a drop-in replacement:

- stop the existing mongod, using the `--shutdown` option as follows:

```
mongod --dbpath /var/mongod/data --shutdown
```

Replace `/var/mongod/data` with your MongoDB dbpath.

- start the new mongod processes with the same dbpath setting, for example:

```
mongod --dbpath /var/mongod/data
```

Replace `/var/mongod/data` with your MongoDB dbpath.

Downgrade to 2.2 After Creating a 2dsphere or text Index If you have created 2dsphere or text indexes while running a 2.4 mongod instance, you can downgrade at any time, by starting the 2.2 mongod with the `--upgrade` option as follows:

```
mongod --dbpath /var/mongod/data/ --upgrade
```

Then, you will need to drop any existing 2dsphere or text indexes using `db.collection.dropIndex()`, for example:

```
db.records.dropIndex( { loc: "2dsphere" } )
db.records.dropIndex( "records_text" )
```

Warning: `--upgrade` will run `repairDatabase` on any database where you have created a `2dsphere` or `text` index, which will rebuild *all* indexes.

Troubleshooting Upgrade/Downgrade Operations If you do not use `--upgrade`, when you attempt to start a 2.2 `mongod` and you have created a `2dsphere` or `text` index, `mongod` will return the following message:

```
'need to upgrade database index_plugin_upgrade with pdfile version 4.6, new version: 4.5 Not upgradin
```

While running 2.4, to check the data file version of a MongoDB database, use the following operation in the shell:

```
db.getSiblingDB('<dbname>').stats().dataFileVersion
```

The major data file ⁶³ version for both 2.2 and 2.4 is 4, the minor data file version for 2.2 is 5 and the minor data file version for 2.4 is 6 **after** you create a `2dsphere` or `text` index.

Compatibility and Index Type Changes in MongoDB 2.4 In 2.4 MongoDB includes two new features related to indexes that users upgrading to version 2.4 must consider, particularly with regard to possible downgrade paths. For more information on downgrades, see [Downgrade MongoDB from 2.4 to Previous Versions](#) (page 626).

New Index Types In 2.4 MongoDB adds two new index types: `2dsphere` and `text`. These index types do not exist in 2.2, and for each database, creating a `2dsphere` or `text` index, will upgrade the data-file version and make that database incompatible with 2.2.

If you intend to downgrade, you should always drop all `2dsphere` and `text` indexes before moving to 2.2.

You can use the [downgrade procedure](#) (page 626) to downgrade these databases and run 2.2 if needed, however this will run a full database repair (as with `repairDatabase`) for all affected databases.

Index Type Validation In MongoDB 2.2 and earlier you could specify invalid index types that did not exist. In these situations, MongoDB would create an ascending (e.g. 1) index. Invalid indexes include index types specified by strings that do not refer to an existing index type, and all numbers other than 1 and -1. ⁶⁴

In 2.4, creating any invalid index will result in an error. Furthermore, you cannot create a `2dsphere` or `text` index on a collection if its containing database has any invalid index types.¹

Example

If you attempt to add an invalid index in MongoDB 2.4, as in the following:

```
db.coll.ensureIndex( { field: "1" } )
```

MongoDB will return the following error document:

```
{
  "err" : "Unknown index plugin '1' in index { field: \"1\" }",
  "code": 16734,
  "n": <number>,
  "connectionId": <number>,
```

⁶³ The data file version (i.e. `pdfile` version) is independent and unrelated to the release version of MongoDB.

⁶⁴ In 2.4, indexes that specify a type of "1" or "-1" (the strings "1" and "-1") will continue to exist, despite a warning on start-up. **However**, a `secondary` in a replica set cannot complete an initial sync from a primary that has a "1" or "-1" index. Avoid all indexes with invalid types.

```
"ok": 1
}
```

See [Upgrade MongoDB to 2.4](#) (page 621) for full upgrade instructions.

Other Resources

- [MongoDB Downloads](#)⁶⁵.
- [All JIRA issues resolved in 2.4](#)⁶⁶.
- [All Backwards incompatible changes](#)⁶⁷.
- [All Third Party License Notices](#)⁶⁸.

See <http://docs.mongodb.org/manual/release-notes/2.4-changes> for an overview of all changes in 2.4.

See also:

See [MongoDB 2.5 Series Development Release Notes](#) for more information about the upcoming release of MongoDB.

11.2 Previous Stable Releases

11.2.1 Release Notes for MongoDB 2.2

See the [full index of this page](#) for a complete list of changes included in 2.2.

- [Upgrading](#) (page 629)
- [Changes](#) (page 631)
- [Licensing Changes](#) (page 638)
- [Resources](#) (page 638)

Upgrading

MongoDB 2.2 is a production release series and succeeds the 2.0 production release series.

MongoDB 2.0 data files are compatible with 2.2-series binaries without any special migration process. However, always perform the upgrade process for replica sets and sharded clusters using the procedures that follow.

Synopsis

- mongod, 2.2 is a drop-in replacement for 2.0 and 1.8.

⁶⁵<http://mongodb.org/downloads>

⁶⁶<https://jira.mongodb.org/secure/IssueNavigator.jspa?reset=true&jqlQuery=project+%3D+SERVER+AND+fixVersion+in+%28%222.3.2%22,%222.3.1%22,%222.4.0-rc1%22,%222.4.0-rc2%22,%222.4.0-rc3%22%29>

⁶⁷<https://jira.mongodb.org/secure/IssueNavigator.jspa?reset=true&jqlQuery=project+%3D+SERVER+AND+fixVersion+in+%28%222.3.2%22%2C+%222.3.1%22%2C+%222.4.0-rc1%22%2C+%222.4.0-rc2%22%2C+%222.4.0-rc3%22%29+AND+%22Backward+Breaking%22+in+%28+Rarely+%2C+sometimes%2C+yes%29>

⁶⁸<https://github.com/mongodb/mongo/blob/v2.4/distsrc/THIRD-PARTY-NOTICES>

- Check your *driver* (page 92) documentation for information regarding required compatibility upgrades, and always run the recent release of your driver.

Typically, only users running with authentication, will need to upgrade drivers before continuing with the upgrade to 2.2.

- For all deployments using authentication, upgrade the drivers (i.e. client libraries), before upgrading the mongod instance or instances.
- For all upgrades of sharded clusters:
 - turn off the balancer during the upgrade process. See the *Disable the Balancer* (page 540) section for more information.
 - upgrade all mongos instances before upgrading any mongod instances.

Other than the above restrictions, 2.2 processes can interoperate with 2.0 and 1.8 tools and processes. You can safely upgrade the mongod and mongos components of a deployment one by one while the deployment is otherwise operational. Be sure to read the detailed upgrade procedures below before upgrading production systems.

Upgrading a Standalone mongod

1. Download binaries of the latest release in the 2.2 series from the [MongoDB Download Page](#)⁶⁹.
2. Shutdown your mongod instance. Replace the existing binary with the 2.2 mongod binary and restart MongoDB.

Upgrading a Replica Set

You can upgrade to 2.2 by performing a “rolling” upgrade of the set by upgrading the members individually while the other members are available to minimize downtime. Use the following procedure:

1. Upgrade the *secondary* members of the set one at a time by shutting down the mongod and replacing the 2.0 binary with the 2.2 binary. After upgrading a mongod instance, wait for the member to recover to SECONDARY state before upgrading the next instance. To check the member’s state, issue `rs.status()` in the mongo shell.
2. Use the mongo shell method `rs.stepDown()` to step down the *primary* to allow the normal *failover* (page 392) procedure. `rs.stepDown()` expedites the failover procedure and is preferable to shutting down the primary directly.

Once the primary has stepped down and another member has assumed PRIMARY state, as observed in the output of `rs.status()`, shut down the previous primary and replace mongod binary with the 2.2 binary and start the new process.

Note: Replica set failover is not instant but will render the set unavailable to read or accept writes until the failover process completes. Typically this takes 10 seconds or more. You may wish to plan the upgrade during a predefined maintenance window.

Upgrading a Sharded Cluster

Use the following procedure to upgrade a sharded cluster:

- *Disable the balancer* (page 540).

⁶⁹<http://downloads.mongodb.org/>

- Upgrade all `mongos` instances *first*, in any order.
- Upgrade all of the `mongod` config server instances using the *stand alone* (page 630) procedure. To keep the cluster online, be sure that at all times at least one config server is up.
- Upgrade each shard’s replica set, using the *upgrade procedure for replica sets* (page 630) detailed above.
- re-enable the balancer.

Note: Balancing is not currently supported in *mixed 2.0.x and 2.2.0* deployments. Thus you will want to reach a consistent version for all shards within a reasonable period of time, e.g. same-day. See SERVER-6902⁷⁰ for more information.

Changes

Major Features

Aggregation Framework The aggregation framework makes it possible to do aggregation operations without needing to use *map-reduce*. The `aggregate` command exposes the aggregation framework, and the `aggregate()` helper in the `mongo` shell provides an interface to these operations. Consider the following resources for background on the aggregation framework and its use:

- Documentation: *Aggregation Concepts* (page 277)
- Reference: *Aggregation Reference* (page 304)
- Examples: *Aggregation Examples* (page 288)

TTL Collections TTL collections remove expired data from a collection, using a special index and a background thread that deletes expired documents every minute. These collections are useful as an alternative to *capped collections* in some cases, such as for data warehousing and caching cases, including: machine generated event data, logs, and session information that needs to persist in a database for only a limited period of time.

For more information, see the *Expire Data from Collections by Setting TTL* (page 160) tutorial.

Concurrency Improvements MongoDB 2.2 increases the server’s capacity for concurrent operations with the following improvements:

1. DB Level Locking⁷¹
2. Improved Yielding on Page Faults⁷²
3. Improved Page Fault Detection on Windows⁷³

To reflect these changes, MongoDB now provides changed and improved reporting for concurrency and use, see `locks` and `server-status-record-stats` in `server status` and see `db.currentOp()`, `mongotop`, and `mongostat`.

Improved Data Center Awareness with Tag Aware Sharding MongoDB 2.2 adds additional support for geographic distribution or other custom partitioning for sharded collections in *clusters*. By using this “tag aware” sharding, you can automatically ensure that data in a sharded database system is always on specific shards. For example, with tag aware sharding, you can ensure that data is closest to the application servers that use that data most frequently.

⁷⁰<https://jira.mongodb.org/browse/SERVER-6902>

⁷¹<https://jira.mongodb.org/browse/SERVER-4328>

⁷²<https://jira.mongodb.org/browse/SERVER-3357>

⁷³<https://jira.mongodb.org/browse/SERVER-4538>

Shard tagging controls data location, and is complementary but separate from replica set tagging, which controls *read preference* (page 401) and *write concern* (page 44). For example, shard tagging can pin all “USA” data to one or more logical shards, while replica set tagging can control which mongod instances (e.g. “production” or “reporting”) the application uses to service requests.

See the documentation for the following helpers in the mongo shell that support tagged sharding configuration:

- sh.addShardTag()
- sh.addTagRange()
- sh.removeShardTag()

Also, see [Tag Aware Sharding](#) (page 547) and [Manage Shard Tags](#) (page 548).

Fully Supported Read Preference Semantics All MongoDB clients and drivers now support full *read preferences* (page 401), including consistent support for a full range of *read preference modes* (page 483) and *tag sets* (page 403). This support extends to the mongos and applies identically to single replica sets and to the replica sets for each shard in a *sharded cluster*.

Additional read preference support now exists in the mongo shell using the `readPref()` cursor method.

Compatibility Changes

Authentication Changes MongoDB 2.2 provides more reliable and robust support for authentication clients, including drivers and mongos instances.

If your cluster runs with authentication:

- For all drivers, use the latest release of your driver and check its release notes.
- In sharded environments, to ensure that your cluster remains available during the upgrade process you **must** use the [upgrade procedure for sharded clusters](#) (page 630).

findAndModify Returns Null Value for Upserts that Perform Inserts In version 2.2, for *upsert* that perform inserts with the `new` option set to `false`, `findAndModify` commands will now return the following output:

```
{ 'ok': 1.0, 'value': null }
```

In the mongo shell, upsert `findAndModify` operations that perform inserts (with `new` set to `false`) only output a `null` value.

In version 2.0 these operations would return an empty document, e.g. `{ }.`

See: [SERVER-6226⁷⁴](#) for more information.

mongodump 2.2 Output Incompatible with Pre-2.2 mongorestore If you use the `mongodump` tool from the 2.2 distribution to create a dump of a database, you must use a 2.2 (or later) version of `mongorestore` to restore that dump.

See: [SERVER-6961⁷⁵](#) for more information.

⁷⁴<https://jira.mongodb.org/browse/SERVER-6226>

⁷⁵<https://jira.mongodb.org/browse/SERVER-6961>

ObjectId().toString() Returns String Literal ObjectId("...") In version 2.2, the `toString()` method returns the string representation of the `ObjectId()` (page 127) object and has the format `ObjectId("...")`.

Consider the following example that calls the `toString()` method on the `ObjectId("507c7f79bcf86cd7994f6c0e")` object:

```
ObjectId("507c7f79bcf86cd7994f6c0e").toString()
```

The method now returns the *string* `ObjectId("507c7f79bcf86cd7994f6c0e")`.

Previously, in version 2.0, the method would return the *hexadecimal string* `507c7f79bcf86cd7994f6c0e`.

If compatibility between versions 2.0 and 2.2 is required, use `ObjectId().str` (page 127), which holds the hexadecimal string value in both versions.

ObjectId().valueOf() Returns hexadecimal string In version 2.2, the `valueOf()` method returns the value of the `ObjectId()` (page 127) object as a lowercase hexadecimal string.

Consider the following example that calls the `valueOf()` method on the `ObjectId("507c7f79bcf86cd7994f6c0e")` object:

```
ObjectId("507c7f79bcf86cd7994f6c0e").valueOf()
```

The method now returns the *hexadecimal string* `507c7f79bcf86cd7994f6c0e`.

Previously, in version 2.0, the method would return the *object* `ObjectId("507c7f79bcf86cd7994f6c0e")`.

If compatibility between versions 2.0 and 2.2 is required, use `ObjectId().str` (page 127) attribute, which holds the hexadecimal string value in both versions.

Behavioral Changes

Restrictions on Collection Names In version 2.2, collection names cannot:

- contain the `$`.
- be an empty string (i.e. `" "`).

This change does not affect collections created with now illegal names in earlier versions of MongoDB.

These new restrictions are in addition to the existing restrictions on collection names which are:

- A collection name should begin with a letter or an underscore.
- A collection name cannot contain the null character.
- Begin with the `system.` prefix. MongoDB reserves `system.` for system collections, such as the `system.indexes` collection.
- The maximum size of a collection name is 128 characters, including the name of the database. However, for maximum flexibility, collections should have names less than 80 characters.

Collections names may have any other valid UTF-8 string.

See the SERVER-4442⁷⁶ and the [Are there any restrictions on the names of Collections?](#) (page 582) FAQ item.

⁷⁶<https://jira.mongodb.org/browse/SERVER-4442>

Restrictions on Database Names for Windows Database names running on Windows can no longer contain the following characters:

/\ . " * < > : | ?

The names of the data files include the database name. If you attempt to upgrade a database instance with one or more of these characters, mongod will refuse to start.

Change the name of these databases before upgrading. See SERVER-4584⁷⁷ and SERVER-6729⁷⁸ for more information.

_id Fields and Indexes on Capped Collections All *capped collections* now have an `_id` field by default, if they exist outside of the `local` database, and now have indexes on the `_id` field. This change only affects capped collections created with 2.2 instances and does not affect existing capped collections.

See: SERVER-5516⁷⁹ for more information.

New \$elemMatch Projection Operator The `$elemMatch` operator allows applications to narrow the data returned from queries so that the query operation will only return the first matching element in an array. See the <http://docs.mongodb.org/manual/reference/operator/projection/elemMatch> documentation and the SERVER-2238⁸⁰ and SERVER-828⁸¹ issues for more information.

Windows Specific Changes

Windows XP is Not Supported As of 2.2, MongoDB does not support Windows XP. Please upgrade to a more recent version of Windows to use the latest releases of MongoDB. See SERVER-5648⁸² for more information.

Service Support for mongos.exe You may now run `mongos.exe` instances as a Windows Service. See the <http://docs.mongodb.org/manual/reference/program/mongos.exe> reference and *MongoDB as a Windows Service* (page 14) and SERVER-1589⁸³ for more information.

Log Rotate Command Support MongoDB for Windows now supports log rotation by way of the `logRotate` database command. See SERVER-2612⁸⁴ for more information.

New Build Using SlimReadWrite Locks for Windows Concurrency Labeled “2008+” on the Downloads Page⁸⁵, this build for 64-bit versions of Windows Server 2008 R2 and for Windows 7 or newer, offers increased performance over the standard 64-bit Windows build of MongoDB. See SERVER-3844⁸⁶ for more information.

⁷⁷<https://jira.mongodb.org/browse/SERVER-4584>

⁷⁸<https://jira.mongodb.org/browse/SERVER-6729>

⁷⁹<https://jira.mongodb.org/browse/SERVER-5516>

⁸⁰<https://jira.mongodb.org/browse/SERVER-2238>

⁸¹<https://jira.mongodb.org/browse/SERVER-828>

⁸²<https://jira.mongodb.org/browse/SERVER-5648>

⁸³<https://jira.mongodb.org/browse/SERVER-1589>

⁸⁴<https://jira.mongodb.org/browse/SERVER-2612>

⁸⁵<http://www.mongodb.org/downloads>

⁸⁶<https://jira.mongodb.org/browse/SERVER-3844>

Tool Improvements

Index Definitions Handled by mongodump and mongorestore When you specify the `--collection` option to `mongodump`, `mongodump` will now backup the definitions for all indexes that exist on the source database. When you attempt to restore this backup with `mongorestore`, the target `mongod` will rebuild all indexes. See SERVER-808⁸⁷ for more information.

`mongorestore` now includes the `--noIndexRestore` option to provide the preceding behavior. Use `--noIndexRestore` to prevent `mongorestore` from building previous indexes.

mongooplog for Replaying Ologs The `mongooplog` tool makes it possible to pull *oplog* entries from `mongod` instance and apply them to another `mongod` instance. You can use `mongooplog` to achieve point-in-time backup of a MongoDB data set. See the SERVER-3873⁸⁸ case and the <http://docs.mongodb.org/manual/reference/program/mongooplog> documentation.

Authentication Support for mongotop and mongostat `mongotop` and `mongostat` now contain support for username/password authentication. See SERVER-3875⁸⁹ and SERVER-3871⁹⁰ for more information regarding this change. Also consider the documentation of the following options for additional information:

- `mongotop --username`
- `mongotop --password`
- `mongostat --username`
- `mongostat --password`

Write Concern Support for mongoimport and mongorestore `mongoimport` now provides an option to halt the import if the operation encounters an error, such as a network interruption, a duplicate key exception, or a write error. The `--stopOnError` option will produce an error rather than silently continue importing data. See SERVER-3937⁹¹ for more information.

In `mongorestore`, the `--w` option provides support for configurable write concern.

mongodump Support for Reading from Secondaries You can now run `mongodump` when connected to a *secondary* member of a *replica set*. See SERVER-3854⁹² for more information.

mongoimport Support for full 16MB Documents Previously, `mongoimport` would only import documents that were less than 4 megabytes in size. This issue is now corrected, and you may use `mongoimport` to import documents that are at least 16 megabytes in size. See SERVER-4593⁹³ for more information.

Timestamp() Extended JSON format MongoDB extended JSON now includes a new `Timestamp()` type to represent the `Timestamp` type that MongoDB uses for timestamps in the *oplog* among other contexts.

This permits tools like `mongooplog` and `mongodump` to query for specific timestamps. Consider the following `mongodump` operation:

⁸⁷<https://jira.mongodb.org/browse/SERVER-808>

⁸⁸<https://jira.mongodb.org/browse/SERVER-3873>

⁸⁹<https://jira.mongodb.org/browse/SERVER-3875>

⁹⁰<https://jira.mongodb.org/browse/SERVER-3871>

⁹¹<https://jira.mongodb.org/browse/SERVER-3937>

⁹²<https://jira.mongodb.org/browse/SERVER-3854>

⁹³<https://jira.mongodb.org/browse/SERVER-4593>

```
mongodump --db local --collection oplog.rs --query '{"ts": {"$gt": {"$timestamp": {"t": 1344969612000}}}}
```

See [SERVER-3483⁹⁴](#) for more information.

Shell Improvements

Improved Shell User Interface 2.2 includes a number of changes that improve the overall quality and consistency of the user interface for the mongo shell:

- Full Unicode support.
- Bash-like line editing features. See [SERVER-4312⁹⁵](#) for more information.
- Multi-line command support in shell history. See [SERVER-3470⁹⁶](#) for more information.
- Windows support for the edit command. See [SERVER-3998⁹⁷](#) for more information.

Helper to load Server-Side Functions The db.loadServerScripts() loads the contents of the current database's system.js collection into the current mongo shell session. See [SERVER-1651⁹⁸](#) for more information.

Support for Bulk Inserts If you pass an array of *documents* to the insert() method, the mongo shell will now perform a bulk insert operation. See [SERVER-3819⁹⁹](#) and [SERVER-2395¹⁰⁰](#) for more information.

Note: For bulk inserts on sharded clusters, the getLastError command alone is insufficient to verify success. Applications should must verify the success of bulk inserts in application logic.

Operations

Support for Logging to Syslog See the [SERVER-2957¹⁰¹](#) case and the documentation of the syslog run-time option or the mongod --syslog and mongos --syslog command line-options.

touch Command Added the touch command to read the data and/or indexes from a collection into memory. See: [SERVER-2023¹⁰²](#) and touch for more information.

indexCounters No Longer Report Sampled Data indexCounters now report actual counters that reflect index use and state. In previous versions, these data were sampled. See [SERVER-5784¹⁰³](#) and indexCounters for more information.

Padding Specifiable on compact Command See the documentation of the compact and the [SERVER-4018¹⁰⁴](#) issue for more information.

⁹⁴<https://jira.mongodb.org/browse/SERVER-3483>

⁹⁵<https://jira.mongodb.org/browse/SERVER-4312>

⁹⁶<https://jira.mongodb.org/browse/SERVER-3470>

⁹⁷<https://jira.mongodb.org/browse/SERVER-3998>

⁹⁸<https://jira.mongodb.org/browse/SERVER-1651>

⁹⁹<https://jira.mongodb.org/browse/SERVER-3819>

¹⁰⁰<https://jira.mongodb.org/browse/SERVER-2395>

¹⁰¹<https://jira.mongodb.org/browse/SERVER-2957>

¹⁰²<https://jira.mongodb.org/browse/SERVER-2023>

¹⁰³<https://jira.mongodb.org/browse/SERVER-5784>

¹⁰⁴<https://jira.mongodb.org/browse/SERVER-4018>

Added Build Flag to Use System Libraries The Boost library, version 1.49, is now embedded in the MongoDB code base.

If you want to build MongoDB binaries using system Boost libraries, you can pass `scons` using the `--use-system-boost` flag, as follows:

```
scons --use-system-boost
```

When building MongoDB, you can also pass `scons` a flag to compile MongoDB using only system libraries rather than the included versions of the libraries. For example:

```
scons --use-system-all
```

See the [SERVER-3829¹⁰⁵](#) and [SERVER-5172¹⁰⁶](#) issues for more information.

Memory Allocator Changed to TCMalloc To improve performance, MongoDB 2.2 uses the TCMalloc memory allocator from Google Perftools. For more information about this change see the [SERVER-188¹⁰⁷](#) and [SERVER-4683¹⁰⁸](#). For more information about TCMalloc, see the documentation of [TCMalloc¹⁰⁹](#) itself.

Replication

Improved Logging for Replica Set Lag When *secondary* members of a replica set fall behind in replication, `mongod` now provides better reporting in the log. This makes it possible to track replication in general and identify what process may produce errors or halt replication. See [SERVER-3575¹¹⁰](#) for more information.

Replica Set Members can Sync from Specific Members The new `replSetSyncFrom` command and new `rs.syncFrom()` helper in the `mongo` shell make it possible for you to manually configure from which member of the set a replica will poll *oplog* entries. Use these commands to override the default selection logic if needed. Always exercise caution with `replSetSyncFrom` when overriding the default behavior.

Replica Set Members will not Sync from Members Without Indexes Unless `buildIndexes: false` To prevent inconsistency between members of replica sets, if the member of a replica set has `buildIndexes` (page 475) set to `true`, other members of the replica set will *not* sync from this member, unless they also have `buildIndexes` (page 475) set to `true`. See [SERVER-4160¹¹¹](#) for more information.

New Option To Configure Index Pre-Fetching during Replication By default, when replicating options, *secondaries* will pre-fetch [Indexes](#) (page 309) associated with a query to improve replication throughput in most cases. The `replIndexPrefetch` setting and `--replIndexPrefetch` option allow administrators to disable this feature or allow the `mongod` to pre-fetch only the index on the `_id` field. See [SERVER-6718¹¹²](#) for more information.

Map Reduce Improvements

In 2.2 Map Reduce received the following improvements:

¹⁰⁵<https://jira.mongodb.org/browse/SERVER-3829>

¹⁰⁶<https://jira.mongodb.org/browse/SERVER-5172>

¹⁰⁷<https://jira.mongodb.org/browse/SERVER-188>

¹⁰⁸<https://jira.mongodb.org/browse/SERVER-4683>

¹⁰⁹<http://goog-perftools.sourceforge.net/doc/tcmalloc.html>

¹¹⁰<https://jira.mongodb.org/browse/SERVER-3575>

¹¹¹<https://jira.mongodb.org/browse/SERVER-4160>

¹¹²<https://jira.mongodb.org/browse/SERVER-6718>

- Improved support for sharded MapReduce¹¹³, and
- MapReduce will retry jobs following a config error¹¹⁴.

Sharding Improvements

Index on Shard Keys Can Now Be a Compound Index If your shard key uses the prefix of an existing index, then you do not need to maintain a separate index for your shard key in addition to your existing index. This index, however, cannot be a multi-key index. See the *Shard Key Indexes* (page 512) documentation and SERVER-1506¹¹⁵ for more information.

Migration Thresholds Modified The *migration thresholds* (page 509) have changed in 2.2 to permit more even distribution of *chunks* in collections that have smaller quantities of data. See the *Migration Thresholds* (page 509) documentation for more information.

Licensing Changes

Added License notice for Google Perftools (TCMalloc Utility). See the *License Notice*¹¹⁶ and the SERVER-4683¹¹⁷ for more information.

Resources

- MongoDB Downloads¹¹⁸.
- All JIRA issues resolved in 2.2¹¹⁹.
- All backwards incompatible changes¹²⁰.
- All third party license notices¹²¹.
- What's New in MongoDB 2.2 Online Conference¹²².

11.2.2 Release Notes for MongoDB 2.0

See the full index of this page for a complete list of changes included in 2.0.

- Upgrading (page 639)
- Changes (page 640)
- Resources (page 644)

¹¹³<https://jira.mongodb.org/browse/SERVER-4521>

¹¹⁴<https://jira.mongodb.org/browse/SERVER-4158>

¹¹⁵<https://jira.mongodb.org/browse/SERVER-1506>

¹¹⁶<https://github.com/mongodb/mongo/blob/v2.2/distsrc/THIRD-PARTY-NOTICES#L231>

¹¹⁷<https://jira.mongodb.org/browse/SERVER-4683>

¹¹⁸<http://mongodb.org/downloads>

¹¹⁹<https://jira.mongodb.org/secure/IssueNavigator.jspa?reset=true&jqlQuery=project+%3D+SERVER+AND+fixVersion+in+%28%222.1.0%22%2C%222.1.1%22%2C%222.2.0-rc1%22%2C%222.2.0-rc2%22%29+ORDER+BY+component+ASC%2C+key+DESC>

¹²⁰<https://jira.mongodb.org/secure/IssueNavigator.jspa?requestId=11225>

¹²¹<https://github.com/mongodb/mongo/blob/v2.2/distsrc/THIRD-PARTY-NOTICES>

¹²²<http://www.mongodb.com/events/webinar/mongodb-online-conference-sept>

Upgrading

Although the major version number has changed, MongoDB 2.0 is a standard, incremental production release and works as a drop-in replacement for MongoDB 1.8.

Preparation

Read through all release notes before upgrading, and ensure that no changes will affect your deployment.

If you create new indexes in 2.0, then downgrading to 1.8 is possible but you must reindex the new collections.

`mongoimport` and `mongoexport` now correctly adhere to the CSV spec for handling CSV input/output. This may break existing import/export workflows that relied on the previous behavior. For more information see SERVER-1097¹²³.

Journaling¹²⁴ is **enabled by default** in 2.0 for 64-bit builds. If you still prefer to run without journaling, start `mongod` with the `--nojournal` run-time option. Otherwise, MongoDB creates journal files during startup. The first time you start `mongod` with journaling, you will see a delay as `mongod` creates new files. In addition, you may see reduced write throughput.

2.0 `mongod` instances are interoperable with 1.8 `mongod` instances; however, for best results, upgrade your deployments using the following procedures:

Upgrading a Standalone `mongod`

1. Download the v2.0.x binaries from the MongoDB Download Page¹²⁵.
2. Shutdown your `mongod` instance. Replace the existing binary with the 2.0.x `mongod` binary and restart MongoDB.

Upgrading a Replica Set

1. Upgrade the *secondary* members of the set one at a time by shutting down the `mongod` and replacing the 1.8 binary with the 2.0.x binary from the MongoDB Download Page¹²⁶.
2. To avoid losing the last few updates on failover you can temporarily halt your application (failover should take less than 10 seconds), or you can set *write concern* (page 44) in your application code to confirm that each update reaches multiple servers.
3. Use the `rs.stepDown()` to step down the primary to allow the normal *failover* (page 392) procedure.

`rs.stepDown()` and `repSetStepDown` provide for shorter and more consistent failover procedures than simply shutting down the primary directly.

When the primary has stepped down, shut down its instance and upgrade by replacing the `mongod` binary with the 2.0.x binary.

Upgrading a Sharded Cluster

1. Upgrade all *config server* instances *first*, in any order. Since config servers use two-phase commit, *shard* configuration metadata updates will halt until all are up and running.

¹²³<https://jira.mongodb.org/browse/SERVER-1097>

¹²⁴<http://www.mongodb.org/display/DOCS/Journaling>

¹²⁵<http://downloads.mongodb.org/>

¹²⁶<http://downloads.mongodb.org/>

2. Upgrade mongos routers in any order.

Changes

Compact Command

A `compact` command is now available for compacting a single collection and its indexes. Previously, the only way to compact was to repair the entire database.

Concurrency Improvements

When going to disk, the server will yield the write lock when writing data that is not likely to be in memory. The initial implementation of this feature now exists:

See SERVER-2563¹²⁷ for more information.

The specific operations yield in 2.0 are:

- Updates by `_id`
- Removes
- Long cursor iterations

Default Stack Size

MongoDB 2.0 reduces the default stack size. This change can reduce total memory usage when there are many (e.g., 1000+) client connections, as there is a thread per connection. While portions of a thread's stack can be swapped out if unused, some operating systems do this slowly enough that it might be an issue. The default stack size is lesser of the system setting or 1MB.

Index Performance Enhancements

v2.0 includes significant improvements to the `index` (page 341). Indexes are often 25% smaller and 25% faster (depends on the use case). When upgrading from previous versions, the benefits of the new index type are realized only if you create a new index or re-index an old one.

Dates are now signed, and the max index key size has increased slightly from 819 to 1024 bytes.

All operations that create a new index will result in a 2.0 index by default. For example:

- Reindexing results on an older-version index results in a 2.0 index. However, reindexing on a secondary does *not* work in versions prior to 2.0. Do not reindex on a secondary. For a workaround, see SERVER-3866¹²⁸.
- The `repairDatabase` command converts indexes to a 2.0 indexes.

To convert all indexes for a given collection to the `2.0 type` (page 640), invoke the `compact` command.

Once you create new indexes, downgrading to 1.8.x will require a re-index of any indexes created using 2.0. See *Build Old Style Indexes* (page 341).

¹²⁷<https://jira.mongodb.org/browse/SERVER-2563>

¹²⁸<https://jira.mongodb.org/browse/SERVER-3866>

Sharding Authentication

Applications can now use authentication with *sharded clusters*.

Replica Sets

Hidden Nodes in Sharded Clusters In 2.0, mongos instances can now determine when a member of a replica set becomes “hidden” without requiring a restart. In 1.8, mongos if you reconfigured a member as hidden, you *had* to restart mongos to prevent queries from reaching the hidden member.

Priorities Each *replica set* member can now have a priority value consisting of a floating-point from 0 to 1000, inclusive. Priorities let you control which member of the set you prefer to have as *primary* the member with the highest priority that can see a majority of the set will be elected primary.

For example, suppose you have a replica set with three members, A, B, and C, and suppose that their priorities are set as follows:

- A’s priority is 2.
- B’s priority is 3.
- C’s priority is 1.

During normal operation, the set will always chose B as primary. If B becomes unavailable, the set will elect A as primary.

For more information, see the [priority](#) (page 476) documentation.

Data-Center Awareness You can now “tag” *replica set* members to indicate their location. You can use these tags to design custom *write rules* (page 44) across data centers, racks, specific servers, or any other architecture choice.

For example, an administrator can define rules such as “very important write” or `customerData` or “audit-trail” to replicate to certain servers, racks, data centers, etc. Then in the application code, the developer would say:

```
db.foo.insert(doc, {w : "very important write"})
```

which would succeed if it fulfilled the conditions the DBA defined for “very important write”.

For more information, see [Tagging](#)¹²⁹.

Drivers may also support tag-aware reads. Instead of specifying `slaveOk`, you specify `slaveOk` with tags indicating which data-centers to read from. For details, see the [MongoDB Drivers and Client Libraries](#) (page 92) documentation.

w : majority You can also set `w` to `majority` to ensure that the write propagates to a majority of nodes, effectively committing it. The value for “majority” will automatically adjust as you add or remove nodes from the set.

For more information, see [Write Concern](#) (page 44).

Reconfiguration with a Minority Up If the majority of servers in a set has been permanently lost, you can now force a reconfiguration of the set to bring it back online.

For more information see [Reconfigure a Replica Set with Unavailable Members](#) (page 450).

¹²⁹<http://www.mongodb.org/display/DOCS/Data+Center+Awareness#DataCenterAwareness-Tagging%28version2.0%29>

Primary Checks for a Caught up Secondary before Stepping Down To minimize time without a *primary*, the `rs.stepDown()` method will now fail if the primary does not see a *secondary* within 10 seconds of its latest optime. You can force the primary to step down anyway, but by default it will return an error message.

See also [Force a Member to Become Primary](#) (page 443).

Extended Shutdown on the Primary to Minimize Interruption When you call the `shutdown` command, the *primary* will refuse to shut down unless there is a *secondary* whose optime is within 10 seconds of the primary. If such a secondary isn't available, the primary will step down and wait up to a minute for the secondary to be fully caught up before shutting down.

Note that to get this behavior, you must issue the `shutdown` command explicitly; sending a signal to the process will not trigger this behavior.

You can also force the primary to shut down, even without an up-to-date secondary available.

Maintenance Mode When `repair` or `compact` runs on a *secondary*, the secondary will automatically drop into “recovering” mode until the operation finishes. This prevents clients from trying to read from it while it's busy.

Geospatial Features

Multi-Location Documents Indexing is now supported on documents which have multiple location objects, embedded either inline or in nested sub-documents. Additional command options are also supported, allowing results to return with not only distance but the location used to generate the distance.

For more information, see [Multi-location Documents](#)¹³⁰.

Polygon searches Polygonal `$within` queries are also now supported for simple polygon shapes. For details, see the `$within` operator documentation.

Journaling Enhancements

- Journaling is now enabled by default for 64-bit platforms. Use the `--nojournal` command line option to disable it.
- The journal is now compressed for faster commits to disk.
- A new `--journalCommitInterval` run-time option exists for specifying your own group commit interval. The default settings do not change.
- A new `{ getLastError: { j: true } }` option is available to wait for the group commit. The group commit will happen sooner when a client is waiting on `{ j: true }`. If journaling is disabled, `{ j: true }` is a no-op.

New `ContinueOnError` Option for Bulk Insert

Set the `continueOnError` option for bulk inserts, in the [driver](#) (page 92), so that bulk insert will continue to insert any remaining documents even if an insert fails, as is the case with duplicate key exceptions or network interruptions. The `getLastError` command will report whether any inserts have failed, not just the last one. If multiple errors occur, the client will only receive the most recent `getLastError` results.

¹³⁰<http://www.mongodb.org/display/DOCS/Geospatial+Indexing#GeospatialIndexing-MultilocationDocuments>

See `OP_INSERT`¹³¹.

Note: For bulk inserts on sharded clusters, the `getLastError` command alone is insufficient to verify success. Applications should must verify the success of bulk inserts in application logic.

Map Reduce

Output to a Sharded Collection Using the new `sharded` flag, it is possible to send the result of a map/reduce to a sharded collection. Combined with the `reduce` or `merge` flags, it is possible to keep adding data to very large collections from map/reduce jobs.

For more information, see [MapReduce Output Options](#)¹³² and <http://docs.mongodb.org/manual/reference/command/mapReduce/>

Performance Improvements Map/reduce performance will benefit from the following:

- Larger in-memory buffer sizes, reducing the amount of disk I/O needed during a job
- Larger javascript heap size, allowing for larger objects and less GC
- Supports pure JavaScript execution with the `jsMode` flag. See <http://docs.mongodb.org/manual/reference/command/mapReduce/>.

New Querying Features

Additional regex options: s Allows the dot (.) to match all characters including new lines. This is in addition to the currently supported `i`, `m` and `x`. See [Regular Expressions](#)¹³³ and `$regex`.

\$and A special boolean `$and` query operator is now available.

Command Output Changes

The output of the `validate` command and the documents in the `system.profile` collection have both been enhanced to return information as BSON objects with keys for each value rather than as free-form strings.

Shell Features

Custom Prompt You can define a custom prompt for the `mongo` shell. You can change the prompt at any time by setting the `prompt` variable to a string or a custom JavaScript function returning a string. For examples, see [Custom Prompt](#)¹³⁴.

Default Shell Init Script On startup, the shell will check for a `.mongorc.js` file in the user's home directory. The shell will execute this file after connecting to the database and before displaying the prompt.

If you would like the shell not to run the `.mongorc.js` file automatically, start the shell with `--norc`.

For more information, see <http://docs.mongodb.org/manual/reference/program/mongo>.

¹³¹<http://www.mongodb.org/display/DOCS/Mongo+Wire+Protocol#MongoWireProtocol-OPINSERT>

¹³²<http://www.mongodb.org/display/DOCS/MapReduce#MapReduce-Outputoptions>

¹³³<http://www.mongodb.org/display/DOCS/Advanced+Queries#AdvancedQueries-RegularExpressions>

¹³⁴<http://www.mongodb.org/display/DOCS/Overview++The+MongoDB+Interactive+Shell#Overview-TheMongoDBInteractiveShell-CustomPrompt>

Most Commands Require Authentication

In 2.0, when running with authentication (e.g. `auth`) *all* database commands require authentication, *except* the following commands.

- `isMaster`
- `authenticate`
- `getnonce`
- `buildInfo`
- `ping`
- `isdbgrid`

Resources

- [MongoDB Downloads](#)¹³⁵
- [All JIRA Issues resolved in 2.0](#)¹³⁶
- [All Backward Incompatible Changes](#)¹³⁷

11.2.3 Release Notes for MongoDB 1.8

See the **full index of this page** for a complete list of changes included in 1.8.

- [Upgrading](#) (page 644)
- [Changes](#) (page 647)
- [Resources](#) (page 650)

Upgrading

MongoDB 1.8 is a standard, incremental production release and works as a drop-in replacement for MongoDB 1.6, except:

- *Replica set* members should be upgraded in a particular order, as described in [Upgrading a Replica Set](#) (page 645).
- The `mapReduce` command has changed in 1.8, causing incompatibility with previous releases. `mapReduce` no longer generates temporary collections (thus, `keepTemp` has been removed). Now, you must always supply a value for `out`. See the `out` field options in the `mapReduce` document. If you use `MapReduce`, this also likely means you need a recent version of your client driver.

Preparation

Read through all release notes before upgrading and ensure that no changes will affect your deployment.

¹³⁵<http://mongodb.org/downloads>

¹³⁶<https://jira.mongodb.org/secure/IssueNavigator.jspa?mode=hide&requestId=11002>

¹³⁷<https://jira.mongodb.org/secure/IssueNavigator.jspa?requestId=11023>

Upgrading a Standalone mongod

1. Download the v1.8.x binaries from the MongoDB Download Page¹³⁸.
2. Shutdown your mongod instance.
3. Replace the existing binary with the 1.8.x mongod binary.
4. Restart MongoDB.

Upgrading a Replica Set

1.8.x *secondaries* can replicate from 1.6.x *primaries*.

1.6.x secondaries cannot replicate from 1.8.x primaries.

Thus, to upgrade a *replica set* you must replace all of your secondaries first, then the primary.

For example, suppose you have a replica set with a primary, an *arbiter* and several secondaries. To upgrade the set, do the following:

1. For the arbiter:
 - (a) Shut down the arbiter.
 - (b) Restart it with the 1.8.x binary from the MongoDB Download Page¹³⁹.
2. Change your config (optional) to prevent election of a new primary.

It is possible that, when you start shutting down members of the set, a new primary will be elected. To prevent this, you can give all of the secondaries a priority of 0 before upgrading, and then change them back afterwards. To do so:

- (a) Record your current config. Run `rs.config()` and paste the results into a text file.
- (b) Update your config so that all secondaries have priority 0. For example:

```
config = rs.conf()
{
  "_id" : "foo",
  "version" : 3,
  "members" : [
    {
      "_id" : 0,
      "host" : "ubuntu:27017"
    },
    {
      "_id" : 1,
      "host" : "ubuntu:27018"
    },
    {
      "_id" : 2,
      "host" : "ubuntu:27019",
      "arbiterOnly" : true
    }
  ],
  "_id" : 3,
  "host" : "ubuntu:27020"
},
```

¹³⁸<http://downloads.mongodb.org/>

¹³⁹<http://downloads.mongodb.org/>

```
        {
            "_id" : 4,
            "host" : "ubuntu:27021"
        },
    ],
}
config.version++
3
rs.isMaster()
{
    "setName" : "foo",
    "ismaster" : false,
    "secondary" : true,
    "hosts" : [
        "ubuntu:27017",
        "ubuntu:27018"
    ],
    "arbiters" : [
        "ubuntu:27019"
    ],
    "primary" : "ubuntu:27018",
    "ok" : 1
}
// for each secondary
config.members[0].priority = 0
config.members[3].priority = 0
config.members[4].priority = 0
rs.reconfig(config)
```

3. For each secondary:
 - (a) Shut down the secondary.
 - (b) Restart it with the 1.8.x binary from the [MongoDB Download Page](#)¹⁴⁰.

4. If you changed the config, change it back to its original state:

```
config = rs.conf()
config.version++
config.members[0].priority = 1
config.members[3].priority = 1
config.members[4].priority = 1
rs.reconfig(config)
```

5. Shut down the primary (the final 1.6 server), and then restart it with the 1.8.x binary from the [MongoDB Download Page](#)¹⁴¹.

Upgrading a Sharded Cluster

1. Turn off the balancer:

```
mongo <a_mongos_hostname>
use config
db.settings.update({_id:"balancer"}, {$set : {stopped:true}}, true)
```

2. For each shard:

¹⁴⁰<http://downloads.mongodb.org/>

¹⁴¹<http://downloads.mongodb.org/>

- If the shard is a *replica set*, follow the directions above for [Upgrading a Replica Set](#) (page 645).
 - If the shard is a single mongod process, shut it down and then restart it with the 1.8.x binary from the [MongoDB Download Page](#)¹⁴².
3. For each mongos:
 - (a) Shut down the mongos process.
 - (b) Restart it with the 1.8.x binary from the [MongoDB Download Page](#)¹⁴³.
 4. For each config server:
 - (a) Shut down the config server process.
 - (b) Restart it with the 1.8.x binary from the [MongoDB Download Page](#)¹⁴⁴.
 5. Turn on the balancer:

```
use config
db.settings.update({ _id: "balancer" }, { $set : { stopped: false } })
```

Returning to 1.6

If for any reason you must move back to 1.6, follow the steps above in reverse. Please be careful that you have not inserted any documents larger than 4MB while running on 1.8 (where the max size has increased to 16MB). If you have you will get errors when the server tries to read those documents.

Journaling Returning to 1.6 after using 1.8 [Journaling](#) (page 229) works fine, as journaling does not change anything about the data file format. Suppose you are running 1.8.x with journaling enabled and you decide to switch back to 1.6. There are two scenarios:

- If you shut down cleanly with 1.8.x, just restart with the 1.6 mongod binary.
- If 1.8.x shut down uncleanly, start 1.8.x up again and let the journal files run to fix any damage (incomplete writes) that may have existed at the crash. Then shut down 1.8.x cleanly and restart with the 1.6 mongod binary.

Changes

Journaling

MongoDB now supports write-ahead [Journaling Mechanics](#) (page 229) to facilitate fast crash recovery and durability in the storage engine. With journaling enabled, a mongod can be quickly restarted following a crash without needing to repair the *collections*. The aggregation framework makes it possible to do aggregation

Sparse and Covered Indexes

[Sparse Indexes](#) (page 331) are indexes that only include documents that contain the fields specified in the index. Documents missing the field will not appear in the index at all. This can significantly reduce index size for indexes of fields that contain only a subset of documents within a *collection*.

[Covered Indexes](#) (page 365) enable MongoDB to answer queries entirely from the index when the query only selects fields that the index contains.

¹⁴²<http://downloads.mongodb.org/>

¹⁴³<http://downloads.mongodb.org/>

¹⁴⁴<http://downloads.mongodb.org/>

Incremental MapReduce Support

The `mapReduce` command supports new options that enable incrementally updating existing *collections*. Previously, a MapReduce job could output either to a temporary collection or to a named permanent collection, which it would overwrite with new data.

You now have several options for the output of your MapReduce jobs:

- You can merge MapReduce output into an existing collection. Output from the Reduce phase will replace existing keys in the output collection if it already exists. Other keys will remain in the collection.
- You can now re-reduce your output with the contents of an existing collection. Each key output by the reduce phase will be reduced with the existing document in the output collection.
- You can replace the existing output collection with the new results of the MapReduce job (equivalent to setting a permanent output collection in previous releases)
- You can compute MapReduce inline and return results to the caller without persisting the results of the job. This is similar to the temporary collections generated in previous releases, except results are limited to 8MB.

For more information, see the `out` field options in the `mapReduce` document.

Additional Changes and Enhancements

1.8.1

- Sharding migrate fix when moving larger chunks.
- Durability fix with background indexing.
- Fixed mongos concurrency issue with many incoming connections.

1.8.0

- All changes from 1.7.x series.

1.7.6

- Bug fixes.

1.7.5

- [Journaling](#) (page 229).
- Extent allocation improvements.
- Improved *replica set* connectivity for mongos.
- `getLastError` improvements for *sharding*.

1.7.4

- mongos routes `slaveOk` queries to *secondaries* in *replica sets*.
- New `mapReduce` output options.
- [Sparse Indexes](#) (page 331).

1.7.3

- Initial *covered index* (page 365) support.
- `Distinct` can use data from indexes when possible.
- `mapReduce` can merge or reduce results into an existing collection.
- `mongod` tracks and `mongostat` displays network usage. See *mongostat*.
- Sharding stability improvements.

1.7.2

- `$rename` operator allows renaming of fields in a document.
- `db.eval()` not to block.
- Geo queries with sharding.
- `mongostat --discover` option
- Chunk splitting enhancements.
- Replica sets network enhancements for servers behind a nat.

1.7.1

- Many sharding performance enhancements.
- Better support for `$elemMatch` on primitives in embedded arrays.
- Query optimizer enhancements on range queries.
- Window service enhancements.
- Replica set setup improvements.
- `$pull` works on primitives in arrays.

1.7.0

- Sharding performance improvements for heavy insert loads.
- Slave delay support for replica sets.
- `getLastErrorDefaults` (page 477) for replica sets.
- Auto completion in the shell.
- Spherical distance for geo search.
- All fixes from 1.6.1 and 1.6.2.

Release Announcement Forum Pages

- [1.8.1¹⁴⁵](#), [1.8.0¹⁴⁶](#)

¹⁴⁵<https://groups.google.com/forum/?fromgroups#!topic/mongodb-user/v09MbHEm62Y>
¹⁴⁶<https://groups.google.com/forum/?fromgroups#!topic/mongodb-user/JeHQOnam6Qk>

- 1.7.6¹⁴⁷, 1.7.5¹⁴⁸, 1.7.4¹⁴⁹, 1.7.3¹⁵⁰, 1.7.2¹⁵¹, 1.7.1¹⁵², 1.7.0¹⁵³

Resources

- MongoDB Downloads¹⁵⁴
- All JIRA Issues resolved in 1.8¹⁵⁵

11.2.4 Release Notes for MongoDB 1.6

See the **full index of this page** for a complete list of changes included in 1.6.

- [Upgrading](#) (page 650)
- [Sharding](#) (page 650)
- [Replica Sets](#) (page 651)
- [Other Improvements](#) (page 651)
- [Installation](#) (page 651)
- [1.6.x Release Notes](#) (page 651)
- [1.5.x Release Notes](#) (page 651)

Upgrading

MongoDB 1.6 is a drop-in replacement for 1.4. To upgrade, simply shutdown mongod then restart with the new binaries.

Please note that you should upgrade to the latest version of whichever driver you're using. Certain drivers, including the Ruby driver, will require the upgrade, and all the drivers will provide extra features for connecting to replica sets.

Sharding

[Sharding](#) (page 487) is now production-ready, making MongoDB horizontally scalable, with no single point of failure. A single instance of mongod can now be upgraded to a distributed cluster with zero downtime when the need arises.

- [Sharding](#) (page 487)
- [Deploy a Sharded Cluster](#) (page 514)
- [Convert a Replica Set to a Replicated Sharded Cluster](#) (page 523)

¹⁴⁷<https://groups.google.com/forum/?fromgroups#!topic/mongodb-user/3t6GNZ1qGYc>

¹⁴⁸<https://groups.google.com/forum/?fromgroups#!topic/mongodb-user/S5R0Tx9wkEg>

¹⁴⁹<https://groups.google.com/forum/?fromgroups#!topic/mongodb-user/9Om3Vuw-y9c>

¹⁵⁰<https://groups.google.com/forum/?fromgroups#!topic/mongodb-user/DfNUrbmflI>

¹⁵¹<https://groups.google.com/forum/?fromgroups#!topic/mongodb-user/df7mwK6Xixo>

¹⁵²<https://groups.google.com/forum/?fromgroups#!topic/mongodb-user/HUR9zYtTpA8>

¹⁵³<https://groups.google.com/forum/?fromgroups#!topic/mongodb-user/TUnJCg9161A>

¹⁵⁴<http://mongodb.org/downloads>

¹⁵⁵<https://jira.mongodb.org/secure/IssueNavigator.jspa?mode=hide&requestId=10172>

Replica Sets

Replica sets (page 373), which provide automated failover among a cluster of n nodes, are also now available.

Please note that replica pairs are now deprecated; we strongly recommend that replica pair users upgrade to replica sets.

- *Replication* (page 373)
- *Deploy a Replica Set* (page 416)
- *Convert a Standalone to a Replica Set* (page 428)

Other Improvements

- The w option (and wtimeout) forces writes to be propagated to n servers before returning success (this works especially well with replica sets)
- \$or queries
- Improved concurrency
- \$slice operator for returning subsets of arrays
- 64 indexes per collection (formerly 40 indexes per collection)
- 64-bit integers can now be represented in the shell using NumberLong
- The findAndModify command now supports upserts. It also allows you to specify fields to return
- \$showDiskLoc option to see disk location of a document
- Support for IPv6 and UNIX domain sockets

Installation

- Windows service improvements
- The C++ client is a separate tarball from the binaries

1.6.x Release Notes

- 1.6.5¹⁵⁶

1.5.x Release Notes

- 1.5.8¹⁵⁷
- 1.5.7¹⁵⁸
- 1.5.6¹⁵⁹
- 1.5.5¹⁶⁰

¹⁵⁶https://groups.google.com/forum/?fromgroups#!topic/mongodb-user/06_QCC05Fpk

¹⁵⁷<https://groups.google.com/forum/?fromgroups#!topic/mongodb-user/uJff1QN6Thk>

¹⁵⁸<https://groups.google.com/forum/?fromgroups#!topic/mongodb-user/OYvz40RWs90>

¹⁵⁹https://groups.google.com/forum/?fromgroups#!topic/mongodb-user/4l0N2U_H0cQ

¹⁶⁰<https://groups.google.com/forum/?fromgroups#!topic/mongodb-user/oO749nvTARY>

- 1.5.4¹⁶¹
- 1.5.3¹⁶²
- 1.5.2¹⁶³
- 1.5.1¹⁶⁴
- 1.5.0¹⁶⁵

You can see a full list of all changes on [JIRA](#)¹⁶⁶.

Thank you everyone for your support and suggestions!

11.2.5 Release Notes for MongoDB 1.4

See the **full index of this page** for a complete list of changes included in 1.4.

- [Upgrading](#) (page 652)
- [Core Server Enhancements](#) (page 652)
- [Replication and Sharding](#) (page 652)
- [Deployment and Production](#) (page 653)
- [Query Language Improvements](#) (page 653)
- [Geo](#) (page 653)

Upgrading

We're pleased to announce the 1.4 release of MongoDB. 1.4 is a drop-in replacement for 1.2. To upgrade you just need to shutdown `mongod`, then restart with the new binaries. (Users upgrading from release 1.0 should review the [1.2 release notes](#) (page 653), in particular the instructions for upgrading the DB format.)

Release 1.4 includes the following improvements over release 1.2:

Core Server Enhancements

- [*concurrency*](#) (page 586) improvements
- indexing memory improvements
- [*background index creation*](#) (page 332)
- better detection of regular expressions so the index can be used in more cases

Replication and Sharding

- better handling for restarting slaves offline for a while
- fast new slaves from snapshots (`--fastsync`)
- configurable slave delay (`--slavedelay`)

¹⁶¹https://groups.google.com/forum/?fromgroups#!topic/mongodb-user/380V_Ec_q1c

¹⁶²<https://groups.google.com/forum/?hl=en&fromgroups#!topic/mongodb-user/hsUQL9CxTQw>

¹⁶³<https://groups.google.com/forum/?fromgroups#!topic/mongodb-user/94EE3HVidAA>

¹⁶⁴<https://groups.google.com/forum/?fromgroups#!topic/mongodb-user/7SBPQ2RsfM>

¹⁶⁵<https://groups.google.com/forum/?fromgroups#!topic/mongodb-user/VAhJcjDGTy0>

¹⁶⁶<https://jira.mongodb.org/secure/IssueNavigator.jspa?mode=hide&requestId=10107>

- replication handles clock skew on master
- `$inc` replication fixes
- sharding alpha 3 - notably 2-phase commit on config servers

Deployment and Production

- *configure “slow threshold” for profiling* (page 171)
- ability to do `fsync + lock` for backing up raw files
- option for separate directory per db (`--directoryperdb`)
- `http://localhost:28017/_status` to get `serverStatus` via http
- REST interface is off by default for security (`--rest` to enable)
- can rotate logs with a db command, `logRotate`
- enhancements to `serverStatus` command (`db.serverStatus()`) - counters and *replication lag* (page 458) stats
- new `mongostat` tool

Query Language Improvements

- `$all` with regex
- `$not`
- partial matching of array elements `$elemMatch`
- `$` operator for updating arrays
- `$addToSet`
- `$unset`
- `$pull` supports object matching
- `$set` with array indexes

Geo

- *2d geospatial search* (page 327)
- geo `$center` and `$box` searches

11.2.6 Release Notes for MongoDB 1.2.x

See the **full index of this page** for a complete list of changes included in 1.2.

- New Features (page 654)
- DB Upgrade Required (page 654)
- Replication Changes (page 654)
- `mongoimport` (page 654)
- field filter changing (page 654)

New Features

- More indexes per collection
- Faster index creation
- Map/Reduce
- Stored JavaScript functions
- Configurable fsync time
- Several small features and fixes

DB Upgrade Required

There are some changes that will require doing an upgrade if your previous version is $\leq 1.0.x$. If you're already using a version $\geq 1.1.x$ then these changes aren't required. There are 2 ways to do it:

- `--upgrade`
 - stop your mongod process
 - run `./mongod --upgrade`
 - start mongod again
- use a slave
 - start a slave on a different port and data directory
 - when its synced, shut down the master, and start the new slave on the regular port.

Ask in the forums or IRC for more help.

Replication Changes

- There have been minor changes in replication. If you are upgrading a master/slave setup from $\leq 1.1.2$ you have to update the slave first.

`mongoimport`

- `mongoimport json` has been removed and is replaced with `mongoimport` that can do json/csv/tsv

field filter changing

- We've changed the semantics of the field filter a little bit. Previously only objects with those fields would be returned. Now the field filter only changes the output, not which objects are returned. If you need that behavior, you can use `$exists`

11.3 Other MongoDB Release Notes

11.3.1 Default Write Concern Change

These release notes outline a change to all driver interfaces released in November 2012. See release notes for specific drivers for additional information.

Changes

As of the releases listed below, there are two major changes to all drivers:

1. All drivers will add a new top-level connection class that will increase consistency for all MongoDB client interfaces.

This change is non-backward breaking: existing connection classes will remain in all drivers for a time, and will continue to operate as expected. However, those previous connection classes are now deprecated as of these releases, and will eventually be removed from the driver interfaces.

The new top-level connection class is named `MongoClient`, or similar depending on how host languages handle namespacing.

2. The default write concern on the new `MongoClient` class will be to acknowledge all write operations ^{[167](#)}. This will allow your application to receive acknowledgment of all write operations.

See the documentation of [Write Concern](#) (page 44) for more information about write concern in MongoDB.

Please migrate to the new `MongoClient` class expeditiously.

Releases

The following driver releases will include the changes outlined in [Changes](#) (page 655). See each driver's release notes for a full account of each release as well as other related driver-specific changes.

- C#, version 1.7
- Java, version 2.10.0
- Node.js, version 1.2
- Perl, version 0.501.1
- PHP, version 1.4
- Python, version 2.4
- Ruby, version 1.8

11.4 MongoDB Version Numbers

For MongoDB 2.4.1, 2.4 refers to the release series and .1 refers to the revision. The second component of the release series (e.g. 4 in 2.4.1) describes the type of release series. Release series ending with even numbers (e.g. 4 above) are *stable* and ready for production, while odd numbers are for *development* and testing only.

Generally, changes in the release series (e.g. 2.2 to 2.4) mark the introduction of new features that may break backwards compatibility. Changes to the revision number mark the release bug fixes and backwards-compatible changes.

Important: Always upgrade to the latest stable revision of your release series.

The version numbering system for MongoDB differs from the system used for the MongoDB drivers. Drivers use only the first number to indicate a major version. For details, see [Driver Version Numbers](#) (page 93).

Example

¹⁶⁷ The drivers will call `getLastError` without arguments, which is logically equivalent to the `w: 1` option; however, this operation allows *replica set* users to override the default write concern with the `getLastErrorDefaults` (page 477) setting in the [Replica Set Configuration](#) (page 474).

Version numbers

- 2.0.0 : Stable release.
 - 2.0.1 : Revision.
 - 2.1.0 : Development release *for testing only*. Includes new features and changes for testing. Interfaces and stability may not be compatible in development releases.
 - 2.2.0 : Stable release. This is a culmination of the 2.1.x development series.
-

About MongoDB Documentation

The MongoDB Manual¹ contains comprehensive documentation on the MongoDB *document*-oriented database management system. This page describes the manual's licensing, editions, and versions, and describes how to make a change request and how to contribute to the manual.

For more information on MongoDB, see [MongoDB: A Document Oriented Database](#)². To download MongoDB, see the [downloads page](#)³.

12.1 License

This manual is licensed under a Creative Commons “Attribution-NonCommercial-ShareAlike 3.0 Unported⁴” (i.e. “CC-BY-NC-SA”) license.

The MongoDB Manual is copyright © 2011-2013 MongoDB, Inc.

12.2 Editions

In addition to the [MongoDB Manual](#)⁵, you can also access this content in the following editions:

- [ePub Format](#)⁶
- [Single HTML Page](#)⁷
- [PDF Format](#)⁸ (without reference.)
- [HTML tar.gz](#)⁹

You also can access PDF files that contain subsets of the MongoDB Manual:

- [MongoDB Reference Manual](#)¹⁰
- [MongoDB CRUD Operations](#)¹¹

¹<http://docs.mongodb.org/manual/#>

²<http://www.mongodb.org/about/>

³<http://www.mongodb.org/downloads>

⁴<http://creativecommons.org/licenses/by-nc-sa/3.0/>

⁵<http://docs.mongodb.org/manual/#>

⁶<http://docs.mongodb.org/v2.4/MongoDB-manual.epub>

⁷<http://docs.mongodb.org/v2.4/single/>

⁸<http://docs.mongodb.org/v2.4/MongoDB-manual.pdf>

⁹<http://docs.mongodb.org/v2.4/manual.tar.gz>

¹⁰<http://docs.mongodb.org/v2.4/MongoDB-reference-manual.pdf>

¹¹<http://docs.mongodb.org/v2.4/MongoDB-crud-guide.pdf>

- Data Models for MongoDB¹²
- MongoDB Data Aggregation¹³
- Replication and MongoDB¹⁴
- Sharding and MongoDB¹⁵
- MongoDB Administration¹⁶
- MongoDB Security¹⁷

MongoDB Reference documentation is also available as part of [dash](#)¹⁸. You can also access the [MongoDB Man Pages](#)¹⁹ which are also distributed with the official MongoDB Packages.

12.3 Version and Revisions

This version of the manual reflects version 2.4 of MongoDB.

See the [MongoDB Documentation Project Page](#)²⁰ for an overview of all editions and output formats of the MongoDB Manual. You can see the full revision history and track ongoing improvements and additions for all versions of the manual from its [GitHub repository](#)²¹.

This edition reflects “v2.4” branch of the documentation as of the “d30958caa561d06f1d1d2c2f5b8bef3e54d6d056” revision. This branch is explicitly accessible via “<http://docs.mongodb.org/v2.4>” and you can always reference the commit of the current manual in the [release.txt](#)²² file.

The most up-to-date, current, and stable version of the manual is always available at “<http://docs.mongodb.org/manual/>”.

12.4 Report an Issue or Make a Change Request

To report an issue with this manual or to make a change request, file a ticket at the [MongoDB DOCS Project on Jira](#)²³.

12.5 Contribute to the Documentation

12.5.1 MongoDB Manual Translation

The original authorship language for all MongoDB documentation is American English. However, ensuring that speakers of other languages can read and understand the documentation is of critical importance to the documentation project.

¹²<http://docs.mongodb.org/v2.4/MongoDB-sharding-guide.pdf>

¹³<http://docs.mongodb.org/v2.4/MongoDB-aggregation-guide.pdf>

¹⁴<http://docs.mongodb.org/v2.4/MongoDB-replication-guide.pdf>

¹⁵<http://docs.mongodb.org/v2.4/MongoDB-sharding-guide.pdf>

¹⁶<http://docs.mongodb.org/v2.4/MongoDB-administration-guide.pdf>

¹⁷<http://docs.mongodb.org/v2.4/MongoDB-security-guide.pdf>

¹⁸<http://kapeli.com/dash>

¹⁹<http://docs.mongodb.org/v2.4/manpages.tar.gz>

²⁰<http://docs.mongodb.org>

²¹<https://github.com/mongodb/docs>

²²<http://docs.mongodb.org/v2.4/release.txt>

²³<https://jira.mongodb.org/browse/DOCS>

In this direction, the MongoDB Documentation project uses the service provided by [Smartling](#)²⁴ to translate the MongoDB documentation into additional non-English languages. This translation project is largely supported by the work of volunteer translators from the MongoDB community who contribute to the translation effort.

If you would like to volunteer to help translate the MongoDB documentation, please:

- complete the [MongoDB Contributor Agreement](#)²⁵, and
- create an account on Smartling at translate.docs.mongodb.org²⁶.

Please use the same email address you use to sign the contributor as you use to create your Smartling account.

The [mongodb-translators](#)²⁷ user group exists to facilitate collaboration between translators and the documentation team at large. You can join the Google Group without signing the contributor's agreement.

We currently have the following languages configured:

- Arabic²⁸
- Chinese²⁹
- Czech³⁰
- French³¹
- German³²
- Hungarian³³
- Indonesian³⁴
- Italian³⁵
- Japanese³⁶
- Korean³⁷
- Lithuanian³⁸
- Polish³⁹
- Portuguese⁴⁰
- Romanian⁴¹
- Russian⁴²
- Spanish⁴³

²⁴<http://smartling.com/>

²⁵<http://www.mongodb.com/legal/contributor-agreement>

²⁶<http://translate.docs.mongodb.org/>

²⁷<http://groups.google.com/group/mongodb-translators>

²⁸<http://ar.docs.mongodb.org>

²⁹<http://cn.docs.mongodb.org>

³⁰<http://cs.docs.mongodb.org>

³¹<http://fr.docs.mongodb.org>

³²<http://de.docs.mongodb.org>

³³<http://hu.docs.mongodb.org>

³⁴<http://id.docs.mongodb.org>

³⁵<http://it.docs.mongodb.org>

³⁶<http://jp.docs.mongodb.org>

³⁷<http://ko.docs.mongodb.org>

³⁸<http://lt.docs.mongodb.org>

³⁹<http://pl.docs.mongodb.org>

⁴⁰<http://pt.docs.mongodb.org>

⁴¹<http://ro.docs.mongodb.org>

⁴²<http://ru.docs.mongodb.org>

⁴³<http://es.docs.mongodb.org>

- [Turkish⁴⁴](#)
- [Ukrainian⁴⁵](#)

If you would like to initiate a translation project to an additional language, please report this issue using the “*Report a Problem*” link above or by posting to the [mongodb-translators⁴⁶](#) list.

Currently the translation project only publishes rendered translation. While the translation effort is currently focused on the web site we are evaluating how to retrieve the translated phrases for use in other media.

See also:

- [Contribute to the Documentation](#) (page 658)
- [Style Guide and Documentation Conventions](#) (page 660)
- [MongoDB Manual Organization](#) (page 668)
- [MongoDB Documentation Practices and Processes](#) (page 664)
- [MongoDB Documentation Build System](#) (page 669)

The entire documentation source for this manual is available in the [mongodb/docs](#) repository⁴⁷, which is one of the MongoDB project repositories on [GitHub⁴⁸](#).

To contribute to the documentation, you can open a [GitHub account⁴⁹](#), fork the [mongodb/docs](#) repository⁵⁰, make a change, and issue a pull request.

In order for the documentation team to accept your change, you must complete the [MongoDB Contributor Agreement⁵¹](#).

You can clone the repository by issuing the following command at your system shell:

```
git clone git://github.com/mongodb/docs.git
```

12.5.2 About the Documentation Process

The MongoDB Manual uses [Sphinx⁵²](#), a sophisticated documentation engine built upon [Python Docutils⁵³](#). The original [reStructured Text⁵⁴](#) files, as well as all necessary Sphinx extensions and build tools, are available in the same repository as the documentation.

For more information on the MongoDB documentation process, see:

Style Guide and Documentation Conventions

This document provides an overview of the style for the MongoDB documentation stored in this repository. The overarching goal of this style guide is to provide an accessible base style to ensure that our documentation is easy to read, simple to use, and straightforward to maintain.

For information regarding the MongoDB Manual organization, see [MongoDB Manual Organization](#) (page 668).

⁴⁴<http://tr.docs.mongodb.org>

⁴⁵<http://uk.docs.mongodb.org>

⁴⁶<http://groups.google.com/group/mongodb-translators>

⁴⁷<https://github.com/mongodb/docs>

⁴⁸<http://github.com/mongodb>

⁴⁹<https://github.com/>

⁵⁰<https://github.com/mongodb/docs>

⁵¹<http://www.mongodb.com/contributor>

⁵²<http://sphinx-doc.org/>

⁵³<http://docutils.sourceforge.net/>

⁵⁴<http://docutils.sourceforge.net/rst.html>

Document History

2011-09-27: Document created with a (very) rough list of style guidelines, conventions, and questions.

2012-01-12: Document revised based on slight shifts in practice, and as part of an effort of making it easier for people outside of the documentation team to contribute to documentation.

2012-03-21: Merged in content from the Jargon, and cleaned up style in light of recent experiences.

2012-08-10: Addition to the “Referencing” section.

2013-02-07: Migrated this document to the manual. Added “map-reduce” terminology convention. Other edits.

Naming Conventions

This section contains guidelines on naming files, sections, documents and other document elements.

- File naming Convention:
 - For Sphinx, all files should have a `.txt` extension.
 - Separate words in file names with hyphens (i.e. `-`.)
 - For most documents, file names should have a terse one or two word name that describes the material covered in the document. Allow the path of the file within the document tree to add some of the required context/categorization. For example it’s acceptable to have `http://docs.mongodb.org/manualcore/sharding.rst` and `http://docs.mongodb.org/manualadministration/sharding.rst`.
 - For tutorials, the full title of the document should be in the file name. For example, `http://docs.mongodb.org/manualtutorial/replace-one-configuration-server-in-a-shard-`
- Phrase headlines and titles so that they the content contained within the section so that users can determine what questions the text will answer, and material that it will address without needing them to read the content. This shortens the amount of time that people spend looking for answers, and improvise search/scanning, and possibly “SEO.”
- Prefer titles and headers in the form of “Using foo” over “How to Foo.”
- When using target references (i.e. `:ref:` references in documents,) use names that include enough context to be intelligible thought all documentations. For example, use “replica-set-secondary-only-node” as opposed to “secondary-only-node”. This is to make the source more usable and easier to maintain.

Style Guide

This includes the local typesetting, English, grammatical, conventions and preferences that all documents in the manual should use. The goal here is to choose good standards, that are clear, and have a stylistic minimalism that does not interfere with or distract from the content. A uniform style will improve user experience, and minimize the effect of a multi-authored document.

Punctuation

- Use the oxford comma.

Oxford commas are the commas in a list of things (e.g. “something, something else, and another thing”) before the conjunction (e.g. “and” or “or.”).
- Do not add two spaces after terminal punctuation, such as periods.

- Place commas and periods inside quotation marks.
- Use title case for headings and document titles. Title case capitalizes the first letter of the first, last, and all significant words.

Verbs Verb tense and mood preferences, with examples:

- **Avoid** the first person. For example do not say, “We will begin the backup process by locking the database,” or “I begin the backup process by locking my database instance.”
- **Use** the second person. “If you need to back up your database, start by locking the database first.” In practice, however, it’s more concise to imply second person using the imperative, as in “Before initiating a backup, lock the database.”
- When indicated, use the imperative mood. For example: “Backup your databases often” and “To prevent data loss, back up your databases.”
- The future perfect is also useful in some cases. For example, “Creating disk snapshots without locking the database will lead to an inconsistent state.”
- Avoid helper verbs, as possible, to increase clarity and concision. For example, attempt to avoid “this does foo” and “this will do foo” when possible. Use “does foo” over “will do foo” in situations where “this foos” is unacceptable.

Referencing

- To refer to future or planned functionality in MongoDB or a driver, *always* link to the Jira case. The Manual’s `conf.py` provides an `:issue:` role that links directly to a Jira case (e.g. `:issue:\`SERVER-9001\``).
- For non-object references (i.e. functions, operators, methods, database commands, settings) always reference only the first occurrence of the reference in a section. You should *always* reference objects, except in section headings.
- Structure references with the *why* first; the link second.

For example, instead of this:

Use the *Convert a Replica Set to a Replicated Sharded Cluster* (page 523) procedure if you have an existing replica set.

Type this:

To deploy a sharded cluster for an existing replica set, see *Convert a Replica Set to a Replicated Sharded Cluster* (page 523).

General Formulations

- Contractions are acceptable insofar as they are necessary to increase readability and flow. Avoid otherwise.
- Make lists grammatically correct.
 - Do not use a period after every item unless the list item completes the unfinished sentence before the list.
 - Use appropriate commas and conjunctions in the list items.
 - Typically begin a bulleted list with an introductory sentence or clause, with a colon or comma.
- The following terms are one word:
 - standalone
 - workflow

- Use “unavailable,” “offline,” or “unreachable” to refer to a mongod instance that cannot be accessed. Do not use the colloquialism “down.”
- Always write out units (e.g. “megabytes”) rather than using abbreviations (e.g. “MB”).

Structural Formulations

- There should be at least two headings at every nesting level. Within an “h2” block, there should be either: no “h3” blocks, 2 “h3” blocks, or more than 2 “h3” blocks.
- Section headers are in title case (capitalize first, last, and all important words) and should effectively describe the contents of the section. In a single document you should strive to have section titles that are not redundant and grammatically consistent with each other.
- Use paragraphs and paragraph breaks to increase clarity and flow. Avoid burying critical information in the middle of long paragraphs. Err on the side of shorter paragraphs.
- Prefer shorter sentences to longer sentences. Use complex formations only as a last resort, if at all (e.g. compound complex structures that require semi-colons).
- Avoid paragraphs that consist of single sentences as they often represent a sentence that has unintentionally become too complex or incomplete. However, sometimes such paragraphs are useful for emphasis, summary, or introductions.

As a corollary, most sections should have multiple paragraphs.

- For longer lists and more complex lists, use bulleted items rather than integrating them inline into a sentence.
- Do not expect that the content of any example (inline or blocked,) will be self explanatory. Even when it feels redundant, make sure that the function and use of every example is clearly described.

ReStructured Text and Typesetting

- Place spaces between nested parentheticals and elements in JavaScript examples. For example, prefer { [a, a, a] } over { [a,a,a] }.
- For underlines associated with headers in RST, use:
 - = for heading level 1 or h1s. Use underlines and overlines for document titles.
 - – for heading level 2 or h2s.
 - ~ for heading level 3 or h3s.
 - ` for heading level 4 or h4s.
- Use hyphens (–) to indicate items of an ordered list.
- Place footnotes and other references, if you use them, at the end of a section rather than the end of a file.

Use the footnote format that includes automatic numbering and a target name for ease of use. For instance a footnote tag may look like: [#note]_ with the corresponding directive holding the body of the footnote that resembles the following: . . . [#note].

Do not include . . . code-block:: [language] in footnotes.

- As it makes sense, use the . . . code-block:: [language] form to insert literal blocks into the text. While the double colon, ::, is functional, the . . . code-block:: [language] form makes the source easier to read and understand.
- For all mentions of referenced types (i.e. commands, operators, expressions, functions, statuses, etc.) use the reference types to ensure uniform formatting and cross-referencing.

Jargon and Common Terms

Database Systems and Processes

- To indicate the entire database system, use “MongoDB,” not mongo or Mongo.
- To indicate the database process or a server instance, use mongod or mongos. Refer to these as “processes” or “instances.” Reserve “database” for referring to a database structure, i.e., the structure that holds collections and refers to a group of files on disk.

Distributed System Terms

- Refer to partitioned systems as “sharded clusters.” Do not use shard clusters or sharded systems.
- Refer to configurations that run with replication as “replica sets” (or “master/slave deployments”) rather than “clusters” or other variants.

Data Structure Terms

- “document” refers to “rows” or “records” in a MongoDB database. Potential confusion with “JSON Documents.”

Do not refer to documents as “objects,” because drivers (and MongoDB) do not preserve the order of fields when fetching data. If the order of objects matter, use an array.

- “field” refers to a “key” or “identifier” of data within a MongoDB document.
- “value” refers to the contents of a “field”.
- “sub-document” describes a nested document.

Other Terms

- Use example.net (and .org or .com if needed) for all examples and samples.
- Hyphenate “map-reduce” in order to avoid ambiguous reference to the command name. Do not camel-case.

Notes on Specific Features

- Geo-Location
 1. While MongoDB *is capable* of storing coordinates in sub-documents, in practice, users should only store coordinates in arrays. (See: DOCS-41⁵⁵.)

MongoDB Documentation Practices and Processes

This document provides an overview of the practices and processes.

⁵⁵<https://jira.mongodb.org/browse/DOCS-41>

Practices

- [Commits \(page 665\)](#)
- [Standards and Practices \(page 665\)](#)
- [Collaboration \(page 665\)](#)
- [Builds \(page 666\)](#)
- [Publication \(page 666\)](#)
- [Branches \(page 666\)](#)
- [Migration from Legacy Documentation \(page 666\)](#)
- [Review Process \(page 667\)](#)
 - [Types of Review \(page 667\)](#)
 - * [Initial Technical Review \(page 667\)](#)
 - * [Content Review \(page 667\)](#)
 - * [Consistency Review \(page 667\)](#)
 - * [Subsequent Technical Review \(page 667\)](#)
 - [Review Methods \(page 667\)](#)

Commits

When relevant, include a Jira case identifier in a commit message. Reference documentation cases when applicable, but feel free to reference other cases from jira.mongodb.org⁵⁶.

Err on the side of creating a larger number of discrete commits rather than bundling large set of changes into one commit.

For the sake of consistency, remove trailing whitespaces in the source file.

“Hard wrap” files to between 72 and 80 characters per-line.

Standards and Practices

- At least two people should vet all non-trivial changes to the documentation before publication. One of the reviewers should have significant technical experience with the material covered in the documentation.
- All development and editorial work should transpire on GitHub branches or forks that editors can then merge into the publication branches.

Collaboration

To propose a change to the documentation, do either of the following:

- Open a ticket in the [documentation project](#)⁵⁷ proposing the change. Someone on the documentation team will make the change and be in contact with you so that you can review the change.
- Using [GitHub](#)⁵⁸, fork the [mongodb/docs repository](#)⁵⁹, commit your changes, and issue a pull request. Someone on the documentation team will review and incorporate your change into the documentation.

⁵⁶<http://jira.mongodb.org/>

⁵⁷<https://jira.mongodb.org/browse/DOCS>

⁵⁸<https://github.com/>

⁵⁹<https://github.com/mongodb/docs>

Builds

Building the documentation is useful because Sphinx⁶⁰ and docutils can catch numerous errors in the format and syntax of the documentation. Additionally, having access to an example documentation as it *will* appear to the users is useful for providing more effective basis for the review process. Besides Sphinx, Pygments, and Python-Docutils, the documentation repository contains all requirements for building the documentation resource.

Talk to someone on the documentation team if you are having problems running builds yourself.

Publication

The makefile for this repository contains targets that automate the publication process. Use `make html` to publish a test build of the documentation in the `build/` directory of your repository. Use `make publish` to build the full contents of the manual from the current branch in the `../public-docs/` directory relative the docs repository.

Other targets include:

- `man` - builds UNIX Manual pages for all Mongodbd utilities.
- `push` - builds and deploys the contents of the `../public-docs/`.
- `pdfs` - builds a PDF version of the manual (requires LaTeX dependencies.)

Branches

This section provides an overview of the git branches in the MongoDB documentation repository and their use.

At the present time, future work transpires in the `master`, with the main publication being `current`. As the documentation stabilizes, the documentation team will begin to maintain branches of the documentation for specific MongoDB releases.

Migration from Legacy Documentation

The MongoDB.org Wiki contains a wealth of information. As the transition to the Manual (i.e. this project and resource) continues, it's *critical* that no information disappears or goes missing. The following process outlines *how* to migrate a wiki page to the manual:

1. Read the relevant sections of the Manual, and see what the new documentation has to offer on a specific topic.
In this process you should follow cross references and gain an understanding of both the underlying information and how the parts of the new content relates its constituent parts.
2. Read the wiki page you wish to redirect, and take note of all of the factual assertions, examples presented by the wiki page.
3. Test the factual assertions of the wiki page to the greatest extent possible. Ensure that example output is accurate. In the case of commands and reference material, make sure that documented options are accurate.
4. Make corrections to the manual page or pages to reflect any missing pieces of information.

The target of the redirect need *not* contain every piece of information on the wiki page, **if** the manual as a whole does, and relevant section(s) with the information from the wiki page are accessible from the target of the redirection.

⁶⁰<http://sphinx.pocoo.org/>

5. As necessary, get these changes reviewed by another writer and/or someone familiar with the area of the information in question.

At this point, update the relevant Jira case with the target that you've chosen for the redirect, and make the ticket unassigned.

6. When someone has reviewed the changes and published those changes to Manual, you, or preferably someone else on the team, should make a final pass at both pages with fresh eyes and then make the redirect.

Steps 1-5 should ensure that no information is lost in the migration, and that the final review in step 6 should be trivial to complete.

Review Process

Types of Review The content in the Manual undergoes many types of review, including the following:

Initial Technical Review Review by an engineer familiar with MongoDB and the topic area of the documentation. This review focuses on technical content, and correctness of the procedures and facts presented, but can improve any aspect of the documentation that may still be lacking. When both the initial technical review and the content review are complete, the piece may be “published.”

Content Review Textual review by another writer to ensure stylistic consistency with the rest of the manual. Depending on the content, this may precede or follow the initial technical review. When both the initial technical review and the content review are complete, the piece may be “published.”

Consistency Review This occurs post-publication and is content focused. The goals of consistency reviews are to increase the internal consistency of the documentation as a whole. Insert relevant cross-references, update the style as needed, and provide background fact-checking.

When possible, consistency reviews should be as systematic as possible and we should avoid encouraging stylistic and information drift by editing only small sections at a time.

Subsequent Technical Review If the documentation needs to be updated following a change in functionality of the server or following the resolution of a user issue, changes may be significant enough to warrant additional technical review. These reviews follow the same form as the “initial technical review,” but is often less involved and covers a smaller area.

Review Methods If you’re not a usual contributor to the documentation and would like to review something, you can submit reviews in any of the following methods:

- If you’re reviewing an open pull request in GitHub, the best way to comment is on the “overview diff,” which you can find by clicking on the “diff” button in the upper left portion of the screen. You can also use the following URL to reach this interface:

[https://github.com/mongodb/docs/pull/\[pull-request-id\]/files](https://github.com/mongodb/docs/pull/[pull-request-id]/files)

Replace [pull-request-id] with the identifier of the pull request. Make all comments inline, using GitHub’s comment system.

You may also provide comments directly on commits, or on the pull request itself but these commit-comments are archived in less coherent ways and generate less useful emails, while comments on the pull request lead to less specific changes to the document.

- Leave feedback on Jira cases in the [DOCS⁶¹](#) project. These are better for more general changes that aren't necessarily tied to a specific line, or affect multiple files.
- Create a fork of the repository in your GitHub account, make any required changes and then create a pull request with your changes.

If you insert lines that begin with any of the following annotations:

```
... TODO:  
TODO:  
... TODO  
TODO
```

followed by your comments, it will be easier for the original writer to locate your comments. The two dots ... format is a comment in reStructured Text, which will hide your comments from Sphinx and publication if you're worried about that.

This format is often easier for reviewers with larger portions of content to review.

MongoDB Manual Organization

This document provides an overview of the global organization of the documentation resource. Refer to the notes below if you are having trouble understanding the reasoning behind a file's current location, or if you want to add new documentation but aren't sure how to integrate it into the existing resource.

If you have questions, don't hesitate to open a ticket in the [Documentation Jira Project⁶²](#) or contact the [documentation team⁶³](#).

Global Organization

Indexes and Experience The documentation project has two "index files": <http://docs.mongodb.org/manualcontents.txt> and <http://docs.mongodb.org/manualindex.txt>. The "contents" file provides the documentation's tree structure, which Sphinx uses to create the left-pane navigational structure, to power the "Next" and "Previous" page functionality, and to provide all overarching outlines of the resource. The "index" file is not included in the "contents" file (and thus builds will produce a warning here) and is the page that users first land on when visiting the resource.

Having separate "contents" and "index" files provides a bit more flexibility with the organization of the resource while also making it possible to customize the primary user experience.

Additionally, in the top level of the source/ directory, there are a number of "topical" index or outline files. These (like the "index" and "contents" files) use the ... `toctree::` directive to provide organization within the documentation. The subject-specific landing pages indexes combine to create the index in the contents file.

Topical Indexes and Meta Organization Because the documentation on any given subject exists in a number of different locations across the resource the "topical" indexes provide the real structure and organization to the resource. This organization makes it possible to provide great flexibility while still maintaining a reasonable organization of files and URLs for the documentation. Consider the following example:

Given that topic such as "replication," has material regarding the administration of replica sets, as well as reference material, an overview of the functionality, and operational tutorials, it makes more sense to include a few locations for documents, and use the meta documents to provide the topic-level organization.

Current landing pages include:

⁶¹<http://jira.mongodb.org/browse/DOCS>

⁶²<https://jira.mongodb.org/browse/DOCS>

⁶³docs@mongodb.com

- administration
- applications
- crud
- faq
- mongo
- reference
- replication
- security
- sharding

Additional topical indexes are forthcoming.

The Top Level Folders

The documentation has a number of top-level folders, that hold all of the content of the resource. Consider the following list and explanations below:

- “administration” - contains all of the operational and architectural information that systems and database administrators need to know in order to run MongoDB. Topics include: monitoring, replica sets, shard clusters, deployment architectures, and configuration.
- “applications” - contains information about application development and use. While most documentation regarding application development is within the purview of the driver documentation, there are some larger topics regarding the use of these features that deserve some coverage in this context. Topics include: drivers, schema design, optimization, replication, and sharding.
- “core” - contains overviews and introduction to the core features, functionality, and concepts of MongoDB. Topics include: replication, sharding, capped collections, journaling/durability, aggregation.
- “reference” - contains references and indexes of shell functions, database commands, status outputs, as well as manual pages for all of the programs come with MongoDB (e.g. mongostat and mongodump.)
- “tutorial” - contains operational guides and tutorials that lead users through common tasks (administrative and conceptual) with MongoDB. This includes programming patterns and operational guides.
- “faq” - contains all the frequently asked questions related to MongoDB, in a collection of topical files.

MongoDB Documentation Build System

This document contains more direct instructions for building the MongoDB documentation.

Getting Started

Install Dependencies The MongoDB Documentation project depends on the following tools:

- GNU Make
- GNU Tar
- Python
- Git

- Sphinx (documentation management toolchain)
- Pygments (syntax highlighting)
- PyYAML (for the generated tables)
- Droopy (Python package for static text analysis)
- Fabric (Python package for scripting and orchestration)
- Inkscape (Image generation.)
- python-argparse (For Python 2.6.)
- LaTeX/PDF LaTeX (typically texlive; for building PDFs)
- Common Utilities (rsync, tar, gzip, sed)

OS X Install Sphinx, Docutils, and their dependencies with `easy_install` the following command:

```
easy_install Sphinx Jinja2 Pygments docutils PyYAML droopy fabric
```

Feel free to use `pip` rather than `easy_install` to install python packages.

To generate the images used in the documentation, [download and install Inkscape](#)⁶⁴.

Optional

To generate PDFs for the full production build, install a TeX distribution (for building the PDF.) If you do not have a LaTeX installation, use [MacTeX](#)⁶⁵. This is **only** required to build PDFs.

Arch Linux Install packages from the system repositories with the following command:

```
pacman -S python2-sphinx python2-yaml inkscape python2-pip
```

Then install the following Python packages:

```
pip install droopy fabric
```

Optional

To generate PDFs for the full production build, install the following packages from the system repository:

```
pacman -S texlive-bin texlive-core texlive-latexextra
```

Debian/Ubuntu Install the required system packages with the following command:

```
apt-get install python-sphinx python-yaml python-argparse inkscape python-pip
```

Then install the following Python packages:

```
pip install droopy fabric
```

Optional

To generate PDFs for the full production build, install the following packages from the system repository:

⁶⁴<http://inkscape.org/download/>

⁶⁵<http://www.tug.org/mactex/2011/>

```
apt-get install texlive-latex-recommended texlive-latex-recommended
```

Setup and Configuration

Clone the repository:

```
git clone git://github.com/mongodb/docs.git
```

Then run the `bootstrap.py` script in the `docs/` repository, to configure the build dependencies:

```
python bootstrap.py
```

This downloads and configures the `mongodb/docs-tools`⁶⁶ repository, which contains the authoritative build system shared between branches of the MongoDB Manual and other MongoDB documentation projects.

You can run `bootstrap.py` regularly to update build system.

Building the Documentation

The MongoDB documentation build system is entirely accessible via `make` targets. For example, to build an HTML version of the documentation issue the following command:

```
make html
```

You can find the build output in `build/<branch>/html`, where `<branch>` is the name of the current branch.

In addition to the `html` target, the build system provides the following targets:

publish Builds and integrates all output for the production build. Build output is in `build/public/<branch>/`. When you run `publish` in the `master`, the build will generate some output in `build/public/`.

push; stage Uploads the production build to the production or staging web servers. Depends on `publish`. Requires access production or staging environment.

push-all; stage-all Uploads the entire content of `build/public/` to the web servers. Depends on `publish`. Not used in common practice.

push-with-delete; stage-with-delete Modifies the action of `push` and `stage` to remove remote file that don't exist in the local build. Use with caution.

html; latex; dirhtml; epub; texinfo; man; json These are standard targets derived from the default Sphinx Makefile, with adjusted dependencies. Additionally, for all of these targets you can append `-nitpick` to increase Sphinx's verbosity, or `-clean` to remove all Sphinx build artifacts.

`latex` performs several additional post-processing steps on `.tex` output generated by Sphinx. This target will also compile PDFs using `pdflatex`.

`html` and `man` also generates a `.tar.gz` file of the build outputs for inclusion in the final releases.

Build Mechanics and Tools

Internally the build system has a number of components and processes. See the `docs-tools` `README`⁶⁷ for more information on the internals. This section documents a few of these components from a very high level and lists useful operations for contributors to the documentation.

⁶⁶<http://github.com/mongodb/docs-tools/>

⁶⁷<https://github.com/mongodb/docs-tools/blob/master/README.rst>

Fabric Fabric is an orchestration and scripting package for Python. The documentation uses Fabric to handle the deployment of the build products to the web servers and also unifies a number of independent build operations. Fabric commands have the following form:

```
fab <module>.<task>[:<argument>]
```

The <argument> is optional in most cases. Additionally some tasks are available at the root level, without a module. To see a full list of fabric tasks, use the following command:

```
fab -l
```

You can chain fabric tasks on a single command line, although this doesn't always make sense.

Important fabric tasks include:

tools.bootstrap Runs the `bootstrap.py` script. Useful for re-initializing the repository without needing to be in root of the repository.

tools.dev; tools.reset `tools.dev` switches the `origin` remote of the `docs-tools` checkout in `build` directory, to `../docs-tools` to facilitate build system testing and development. `tools.reset` resets the `origin` remote for normal operation.

tools.conf `tools.conf` returns the content of the configuration object for the current project. These data are useful during development.

stats.report:<filename> Returns, a collection of readability statistics. Specify file names relative to source/ tree.

make Provides a thin wrapper around Make calls. Allows you to start make builds from different locations in the project repository.

process.refresh_dependencies Updates the time stamp of `.txt` source files with changed include files, to facilitate Sphinx's incremental rebuild process. This task runs internally as part of the build process.

Buildcloth `Buildcloth`⁶⁸ is a meta-build tool, used to generate Makefiles programatically. This makes the build system easier to maintain, and makes it easier to use the same fundamental code to generate various branches of the Manual as well as related documentation projects. See [makecloth/ in the docs-tools repository](#)⁶⁹ for the relevant code.

Running `make` with no arguments will regenerate these parts of the build system automatically.

Rstcloth `Rstcloth`⁷⁰ is a library for generating reStructuredText programatically. This makes it possible to generate content for the documentation, such as tables, tables of contents, and API reference material programatically and transparently. See [rstcloth/ in the docs-tools repository](#)⁷¹ for the relevant code.

If you have any questions, please feel free to open a [Jira Case](#)⁷².

⁶⁸<https://pypi.python.org/pypi/buildcloth/>

⁶⁹<https://github.com/mongodb/docs-tools/tree/master/makecloth>

⁷⁰<https://pypi.python.org/pypi/rstcloth>

⁷¹<https://github.com/mongodb/docs-tools/tree/master/rstcloth>

⁷²<https://jira.mongodb.org/browse/DOCS>

Symbols

_id, 316
_id index, 316
<database>.system.indexes (MongoDB reporting output), 223
<database>.system.js (MongoDB reporting output), 223
<database>.system.namespaces (MongoDB reporting output), 223
<database>.system.profile (MongoDB reporting output), 223
<database>.system.users (MongoDB reporting output), 268
<database>.system.users.pwd (MongoDB reporting output), 269
<database>.system.users.roles (MongoDB reporting output), 269
<database>.system.users.user (MongoDB reporting output), 269
<database>.system.users.userSource (MongoDB reporting output), 269
0 (error code), 230
100 (error code), 231
12 (error code), 231
14 (error code), 231
2 (error code), 230
20 (error code), 231
3 (error code), 230
4 (error code), 231
45 (error code), 231
47 (error code), 231
48 (error code), 231
49 (error code), 231
5 (error code), 231

A

addShard (database command), 566
admin.system.users.otherDBRoles (MongoDB reporting output), 269
administration tutorials, 181
ARBITER (replica set state), 481

B

balancing, 508
configure, 537
internals, 508
migration, 509
operations, 538
secondary throttle, 538

C

chunks._id (MongoDB reporting output), 125
chunks.data (MongoDB reporting output), 126
chunks.files_id (MongoDB reporting output), 126
chunks.n (MongoDB reporting output), 126
clusterAdmin (user role), 265
collection
 system, 223
compound index, 318
config, 496
config (MongoDB reporting output), 556
config databases, 496
config servers, 495
config.changelog (MongoDB reporting output), 556
config.changelog._id (MongoDB reporting output), 557
config.changelog.clientAddr (MongoDB reporting output), 557
config.changelog.details (MongoDB reporting output), 557
config.changelog.ns (MongoDB reporting output), 557
config.changelog.server (MongoDB reporting output), 557
config.changelog.time (MongoDB reporting output), 557
config.changelog.what (MongoDB reporting output), 557
config.chunks (MongoDB reporting output), 557
config.collections (MongoDB reporting output), 558
config.databases (MongoDB reporting output), 558
config.lockpings (MongoDB reporting output), 558
config.locks (MongoDB reporting output), 558
config.mongos (MongoDB reporting output), 559
config.settings (MongoDB reporting output), 559
config.shards (MongoDB reporting output), 560

config.tags (MongoDB reporting output), 560
config.version (MongoDB reporting output), 560
connection pooling
 read operations, 36
consistency
 rollbacks, 397
crud
 write operations, 40

D

data_binary (BSON type), 225
data_date (BSON type), 225
data_maxkey (BSON type), 225
data_minkey (BSON type), 225
data_oid (BSON type), 225
data_ref (BSON type), 225
data_regex (BSON type), 225
data_timestamp (BSON type), 225
data_undefined (BSON type), 225
database, 496
 local, 479
database references, 122
db.isMaster (shell method), 463, 467
dbAdmin (user role), 264
dbAdminAnyDatabase (user role), 267
DBRef, 122
development tutorials, 183
DOWN (replica set state), 482

E

EDITOR, 586
enableSharding (database command), 567
environment variable
 EDITOR, 586
 HOME, 209

F

failover
 replica set, 392
FATAL (replica set state), 482
files._id (MongoDB reporting output), 126
files.aliases (MongoDB reporting output), 126
files.chunkSize (MongoDB reporting output), 126
files.contentType (MongoDB reporting output), 126
files.filename (MongoDB reporting output), 126
files.length (MongoDB reporting output), 126
files.md5 (MongoDB reporting output), 126
files.metadata (MongoDB reporting output), 126
files.uploadDate (MongoDB reporting output), 126
fundamentals
 sharding, 497

G

geospatial queries, 350

exact, 350
GridFS, 102, 125
 chunks collection, 125
 collections, 125
 files collection, 126
 index, 103
 initialize, 102

H

HOME, 209

I

index
 _id, 316
 background creation, 332
 compound, 318, 336
 create, 335, 336
 create in background, 341
 drop duplicates, 333, 337
 duplicates, 333, 337
 embedded fields, 317
 hashed, 329, 338
 list indexes, 344
 measure use, 344
 monitor index building, 343
 multikey, 320
 name, 334
 options, 332
 overview, 309
 rebuild, 343
 remove, 342
 replica set, 339
 sort order, 319
 sparse, 331, 338
 subdocuments, 317
 TTL index, 330
 unique, 330, 337
index types, 315
 primary key, 316
installation, 3
installation guides, 3
installation tutorials, 3
internals
 config database, 555
isMaster (database command), 467
isMaster.arbiterOnly (MongoDB reporting output), 468
isMaster.arbiters (MongoDB reporting output), 468
isMaster.hidden (MongoDB reporting output), 469
isMaster.hosts (MongoDB reporting output), 468
isMaster.ismaster (MongoDB reporting output), 467
isMaster.localTime (MongoDB reporting output), 468
isMaster.maxBsonObjectSize (MongoDB reporting output), 468

isMaster.maxMessageSizeBytes (MongoDB reporting output), 468
 isMaster.me (MongoDB reporting output), 469
 isMaster.msg (MongoDB reporting output), 468
 isMaster.passive (MongoDB reporting output), 468
 isMaster.passives (MongoDB reporting output), 468
 isMaster.primary (MongoDB reporting output), 468
 isMaster.secondary (MongoDB reporting output), 468
 isMaster.setName (MongoDB reporting output), 468
 isMaster.tags (MongoDB reporting output), 469

L

listShards (database command), 567
 local database, 479
 local.oplog.\$main (MongoDB reporting output), 481
 local.oplog.rs (MongoDB reporting output), 480
 local.replset.minvalid (MongoDB reporting output), 480
 local.slaves (MongoDB reporting output), 480, 481
 local.sources (MongoDB reporting output), 481
 local.startup_log (MongoDB reporting output), 479
 local.startup_log._id (MongoDB reporting output), 480
 local.startup_log.buildinfo (MongoDB reporting output),
 480
 local.startup_log.cmdLine (MongoDB reporting output),
 480
 local.startup_log.hostname (MongoDB reporting output),
 480
 local.startup_log.pid (MongoDB reporting output), 480
 local.startup_log.startTime (MongoDB reporting output),
 480
 local.startup_log.startTimeLocal (MongoDB reporting output), 480
 local.system.replset (MongoDB reporting output), 480
 local.system.replset._id (MongoDB reporting output),
 474
 local.system.replset.members (MongoDB reporting output), 474
 local.system.replset.members[n]._id (MongoDB reporting output), 474
 local.system.replset.members[n].arbiterOnly (MongoDB reporting output), 475
 local.system.replset.members[n].buildIndexes (MongoDB reporting output), 475
 local.system.replset.members[n].hidden (MongoDB reporting output), 475
 local.system.replset.members[n].host (MongoDB reporting output), 475
 local.system.replset.members[n].priority (MongoDB reporting output), 476
 local.system.replset.members[n].slaveDelay (MongoDB reporting output), 476
 local.system.replset.members[n].tags (MongoDB reporting output), 476

local.system.replset.members[n].votes (MongoDB reporting output), 476
 local.system.replset.settings (MongoDB reporting output), 477
 local.system.replset.settings.chainingAllowed (MongoDB reporting output), 477
 local.system.replset.settings.getLastErrorDefaults (MongoDB reporting output), 477
 local.system.replset.settings.getLastErrorModes (MongoDB reporting output), 477

M

mongos, 503

N

namespace
 local, 479
 system, 223
 nearest (read preference mode), 484

P

primary (read preference mode), 483
 PRIMARY (replica set state), 481
 primaryPreferred (read preference mode), 483

Q

query optimizer, 35

R

read (user role), 263
 read operation
 architecture, 36
 connection pooling, 36
 read operations
 query, 29
 read preference, 401
 background, 401
 behavior, 404
 member selection, 404
 modes, 483
 mongos, 405
 nearest, 404
 ping time, 404
 semantics, 483
 sharding, 405
 tag sets, 403, 446
 readAnyDatabase (user role), 267
 readWrite (user role), 264
 readWriteAnyDatabase (user role), 267
 RECOVERING (replica set state), 482
 references, 122
 removeShard (database command), 569
 replica set

elections, 393
failover, 392, 393
index, 339
local database, 479
network partitions, 393
reconfiguration, 450
resync, 445, 446
rollbacks, 397
security, 236
sync, 406, 445
tag sets, 446
replica set members
 arbiters, 384
 delayed, 383
 hidden, 383
 non-voting, 396
repSetFreeze (database command), 469
repSetGetStatus (database command), 469
repSetGetStatus.date (MongoDB reporting output), 470
repSetGetStatus.members (MongoDB reporting output),
 470
repSetGetStatus.members errmsg (MongoDB reporting
 output), 470
repSetGetStatus.members.health (MongoDB reporting
 output), 470
repSetGetStatus.members.lastHeartbeat (MongoDB re-
 porting output), 470
repSetGetStatus.members.name (MongoDB reporting
 output), 470
repSetGetStatus.members.optime (MongoDB reporting
 output), 470
repSetGetStatus.members.optime.i (MongoDB reporting
 output), 470
repSetGetStatus.members.optime.t (MongoDB reporting
 output), 470
repSetGetStatus.members.optimeDate (MongoDB re-
 porting output), 470
repSetGetStatus.members.pingMS (MongoDB reporting
 output), 471
repSetGetStatus.members.self (MongoDB reporting out-
 put), 470
repSetGetStatus.members.state (MongoDB reporting
 output), 470
repSetGetStatus.members.stateStr (MongoDB reporting
 output), 470
repSetGetStatus.members.uptime (MongoDB reporting
 output), 470
repSetGetStatus.myState (MongoDB reporting output),
 470
repSetGetStatus.set (MongoDB reporting output), 470
repSetGetStatus.syncingTo (MongoDB reporting out-
 put), 471
repSetInitiate (database command), 471
repSetMaintenance (database command), 471
repSetReconfig (database command), 472
repSetSyncFrom (database command), 473
resync (database command), 469
ROLLBACK (replica set state), 482
rollbacks, 397
rs.add (shell method), 465
rs.addArb (shell method), 465
rs.conf (shell method), 464
rs.config (shell method), 464
rs.freeze (shell method), 466
rs.help (shell method), 467
rs.initiate (shell method), 463
rs.reconfig (shell method), 464
rs.remove (shell method), 466
rs.slaveOk (shell method), 466
rs.status (shell method), 463
rs.stepDown (shell method), 466
rs.syncFrom (shell method), 467

S

secondary (read preference mode), 483
SECONDARY (replica set state), 481
secondary throttle, 538
secondaryPreferred (read preference mode), 484
security
 replica set, 236
sh.addShard (shell method), 561
sh.addShardTag (shell method), 565
sh.addTagRange (shell method), 565
sh.enableSharding (shell method), 561
sh.help (shell method), 566
sh.isBalancerRunning (shell method), 564
sh.moveChunk (shell method), 563
sh.removeShardTag (shell method), 566
sh.setBalancerState (shell method), 563
sh.shardCollection (shell method), 562
sh.splitAt (shell method), 563
sh.splitFind (shell method), 562
sh.status (shell method), 564
shard key, 499
 cardinality, 519
 query isolation, 501
 write scaling, 500
shardCollection (database command), 567
sharded clusters, 513
sharding
 chunk size, 511
 config database, 555
 config servers, 495
 localhost, 237
 shard key, 499
 shard key indexes, 512
 shards, 493
shardingState (database command), 568

shards, 493
 SHUNNED (replica set state), 482
 slaveOk, 401
 STARTUP (replica set state), 482
 STARTUP2 (replica set state), 482
 system
 collections, 223
 namespace, 223
 system.profile.client (MongoDB reporting output), 229
 system.profile.command (MongoDB reporting output),
 227
 system.profile.cursorid (MongoDB reporting output), 227
 system.profile.keyUpdates (MongoDB reporting output),
 228
 system.profile.lockStats (MongoDB reporting output),
 228
 system.profile.lockStats.timeAcquiringMicros (MongoDB reporting output), 228
 system.profile.lockStats.timeLockedMicros (MongoDB reporting output), 228
 system.profile.millis (MongoDB reporting output), 229
 system.profile.moved (MongoDB reporting output), 227
 system.profile.nmoved (MongoDB reporting output), 228
 system.profile.nreturned (MongoDB reporting output),
 228
 system.profile.ns (MongoDB reporting output), 227
 system.profile.nscanned (MongoDB reporting output),
 227
 system.profile.ntoreturn (MongoDB reporting output),
 227
 system.profile.ntoskip (MongoDB reporting output), 227
 system.profile.numYield (MongoDB reporting output),
 228
 system.profile.nupdated (MongoDB reporting output),
 228
 system.profile.op (MongoDB reporting output), 227
 system.profile.query (MongoDB reporting output), 227
 system.profile.responseLength (MongoDB reporting output), 228
 system.profile.ts (MongoDB reporting output), 227
 system.profile.updateobj (MongoDB reporting output),
 227
 system.profile.user (MongoDB reporting output), 229

T

tag sets, 403
 configuration, 446
 text search tutorials, 184
 TTL index, 330
 tutorials, 181
 administration, 181
 development patterns, 183
 installation, 3
 text search, 184

U

UNKNOWN (replica set state), 482
 userAdmin (user role), 265
 userAdminAnyDatabase (user role), 267

W

write concern, 44
 write operations, 40